AFIPS

CONFERENCE PROCEEDINGS

VOLUME 41
PART II

1972

FALL JOINT COMPUTER CONFERENCE

December 5 - 7, 1972
Anaheim, California

AFIPS PRESS
210 SUMMIT AVENUE
MONTVALE, NEW JERSEY 07645

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1972 Fall Joint Computer Conference Committee or the American Federation of Information Processing Societies, Inc.

Library of Congress Catalog Card Number 55-44701

AFIPS PRESS
210 Summit Avenue
Montvale, New Jersey 07645

Printed in the United States of America

CONTENTS

PART II

MEASUREMENT OF COMPUTER SYSTEMS—EXECUTIVE VIEWPOINT

ARCHITECTURE—TOPICS OF GENERAL INTEREST

DISTRIBUTED COMPUTING AND NETWORKS

NATURAL LANGUAGE PROCESSING

MEASUREMENT OF COMPUTER SYSTEMS—SOFTWARE VALIDATION AND RELIABILITY

COMPUTER AIDED DESIGN

COMPUTER NETWORK MANAGEMENT

SYSTEMS FOR PROGRAMMING

MEASUREMENT OF COMPUTER SYSTEMS—MONITORS AND THEIR APPLICATIONS

HISTORICAL PERSPECTIVES

INTERACTIVE PROCESSING—EXPERIENCES AND POSSIBILITIES

IMPACT OF NEW TECHNOLOGY ON ARCHITECTURE

ROBOTICS AND TELEOPERATORS

DATA MANAGEMENT SYSTEMS

MEASUREMENT OF COMPUTER SYSTEMS—ANALYTICAL CONSIDERATIONS

TECHNOLOGY AND ARCHITECTURE

LANGUAGE FOR ARTIFICIAL INTELLIGENCE

USER REQUIREMENTS OF AN INFORMATION SYSTEM

MEASUREMENT OF COMPUTER SYSTEMS—CASE STUDIES

SERVICE ASPECTS OF COMMUNICATIONS FOR REMOTE COMPUTING

TRAINING APPLICATIONS FOR VARIOUS GROUPS OF COMPUTER PERSONNEL

ADVANCED TECHNICAL DEVICES

ADVANCES IN NUMERICAL COMPUTATION

Cognitive and creative test generators

by F. D. VICKERS

University of Florida
Gainesville, Florida

INTRODUCTION

No one in education would deny the desirability of being able to produce quizzes and tests by machine. If one is careful and mechanically inclined, a teacher can build up, over a period of time, a bank of questions which can be used in a computer aided test production system. Questions can be drawn from the question (or item) bank on various bases such as random, subject area, level of difficulty, type of question, behavioral objective, or other pertinent characteristic. However, such an item bank requires constant maintenance and new questions should periodically be added.

It is the intention of this paper to demonstrate a more general approach, one that may require more initial effort but in the long run should almost eliminate the need to compose additional questions unless the subject material covered changes or the course objectives change. This approach involves the design and implementation of a computer program that generates a set of questions, or question elements, on a guided but random basis using a set of predetermined question models. Here the word *generate* is used in a different sense from that used in item banking systems. The approach described here involves a system that creates questions from an item bank which is, for all practical purposes, of infinite size yet does not require a great deal of storage space. Storage is primarily devoted to the program.

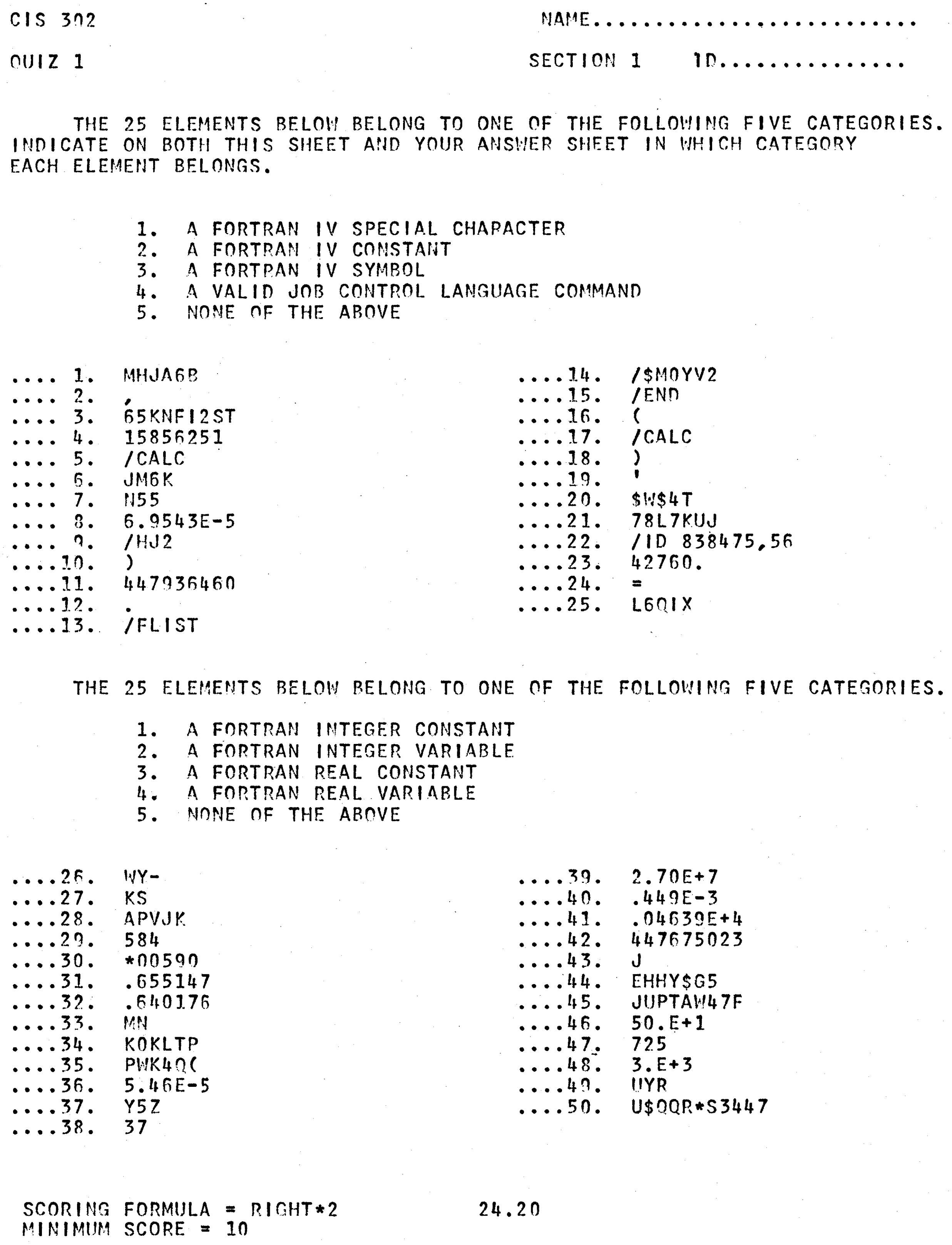
It appears at this stage of our research that this approach would only be applicable to subject material which obeys a set of laws involving quantifiable parameters. However, these quantities need not be purely numerical as the following discussion will demonstrate. The subject area currently being partially tested with this approach is the Fortran language and its usage.

The following section of this paper presents a brief summary of a relatively simple concept which has yielded a useful generator for a particular type of test question. This presentation provides background material for the discussion of concepts which are not so simple and which are now under investigation. Finally, the last section provides some ideas for future development.

SYNTAX QUESTION GENERATION

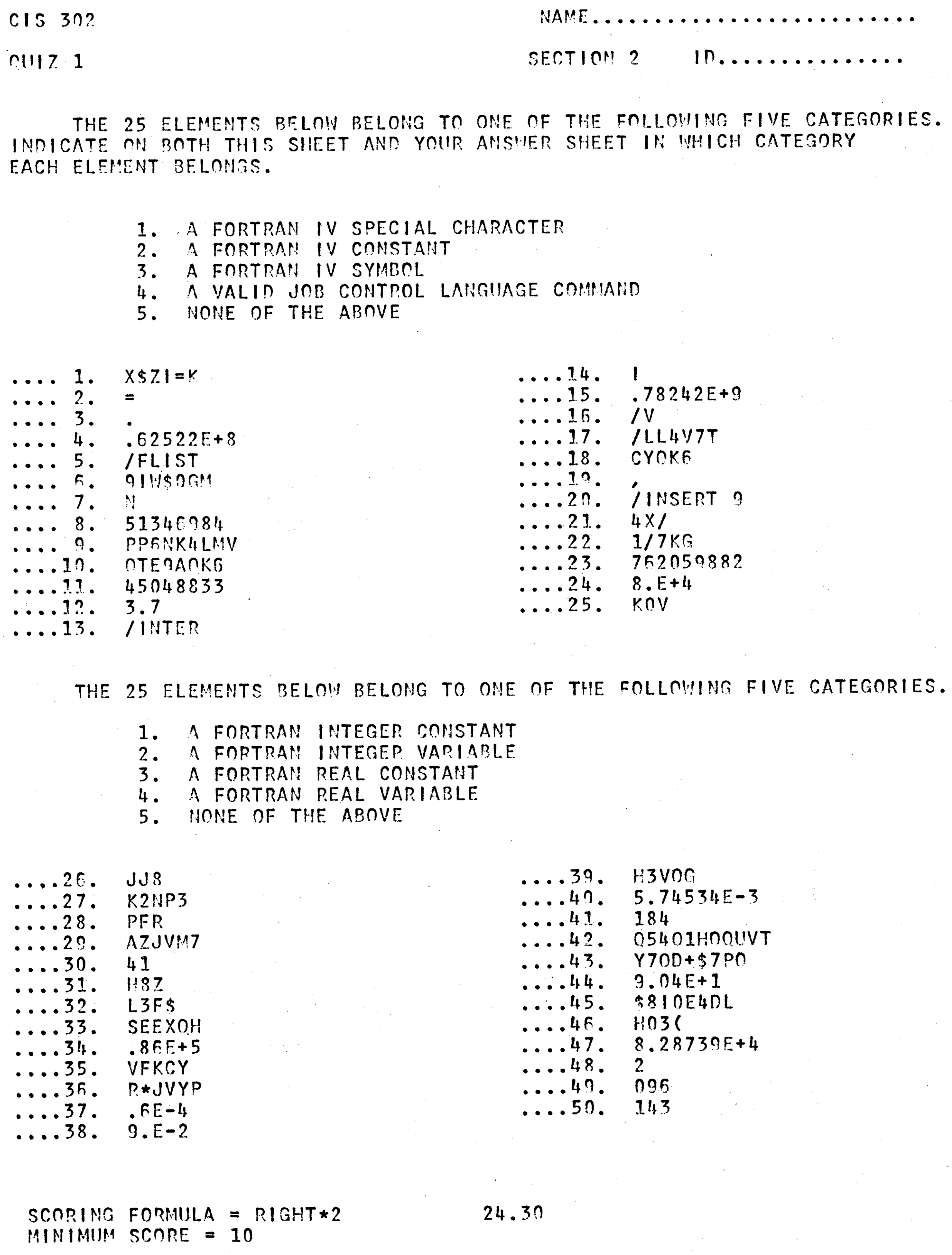
A computer program has been in use at the University of Florida for over six years that generates a set of quizzes composed of questions concerning the syntax of Fortran language elements. See Figures 1 through 5. The student must discriminate between each syntactic type of element as well as invalid constructions. The program is capable of producing quizzes on four different sets of subject area as well as any number of variations within each area. Thus a different variation of a quiz can be produced for each section of the course. Figure 2 contains such a variation of the quiz shown in Figure 1. The only change required in the computer program to obtain the variation is to provide a single different value on input which becomes the seed of a psuedo random number generator. With a different seed a different sequence of random numbers is produced thereby generating different variations of question elements.

For each question, the program first generates a random integer between 1 and 5 to determine the answer category in which to generate the element. As an example, consider Question 27 in Figure 1. The random integer in this case was 2 thus a Fortran integer variable name had to be created for this question. A call was made to a subroutine which proceeds to generate the required name. This subroutine first obtains a random integer between 1 and 6 which represents the length of the name. For Question 27, the result was a 2. The routine then enters a loop to generate each character in the name. Since for integer names the first character must be I, J, K, L, M or N,



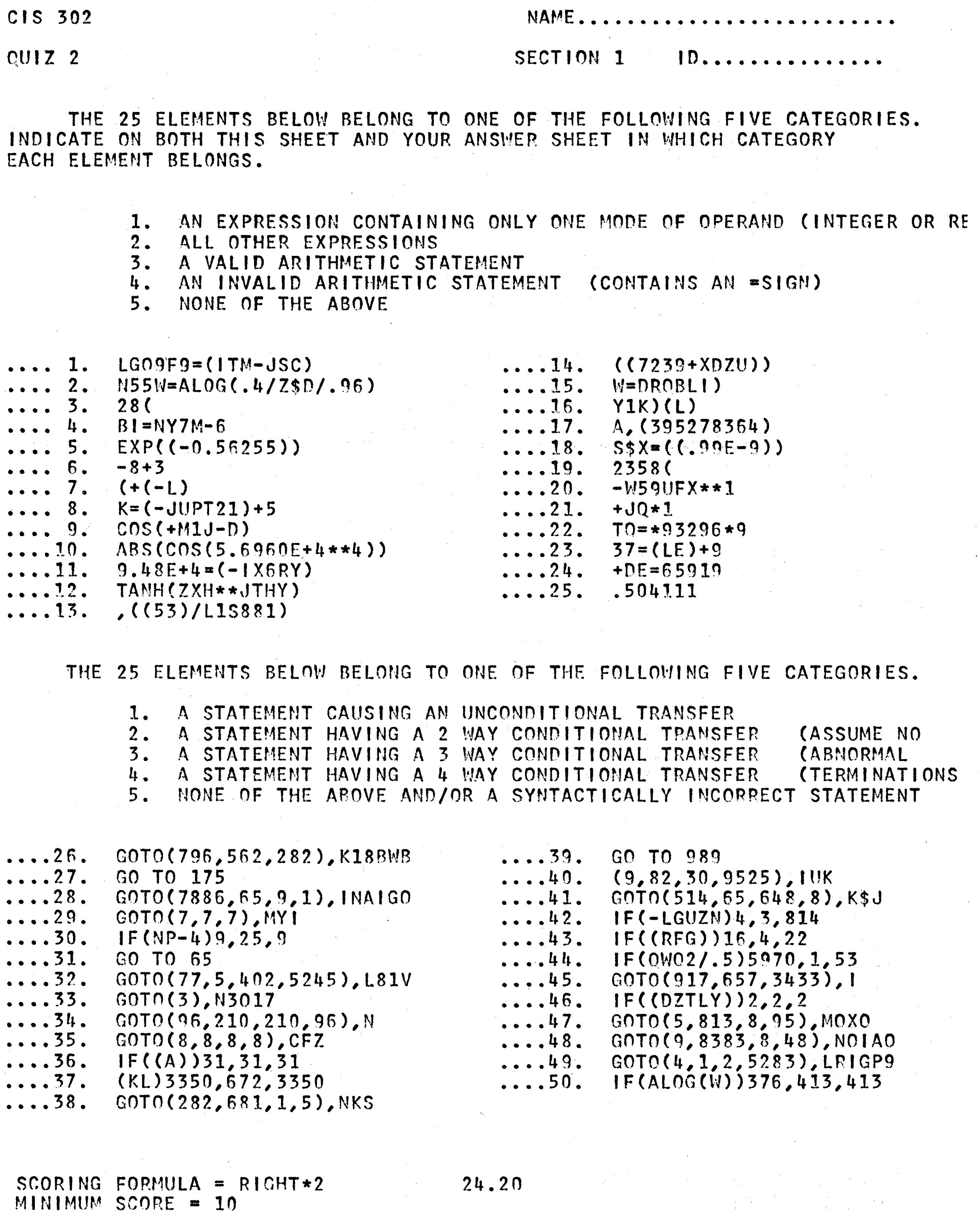
```
CIS 302                                          NAME..........................

QUIZ 1                                          SECTION 1    ID...............

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.
INDICATE ON BOTH THIS SHEET AND YOUR ANSWER SHEET IN WHICH CATEGORY
EACH ELEMENT BELONGS.

          1.  A FORTRAN IV SPECIAL CHARACTER
          2.  A FORTRAN IV CONSTANT
          3.  A FORTRAN IV SYMBOL
          4.  A VALID JOB CONTROL LANGUAGE COMMAND
          5.  NONE OF THE ABOVE

.... 1.  MHJA6B                              ....14.  /$MOYV2
.... 2.  ,                                   ....15.  /END
.... 3.  65KNFI2ST                           ....16.  (
.... 4.  15856251                            ....17.  /CALC
.... 5.  /CALC                               ....18.  )
.... 6.  JM6K                                ....19.  '
.... 7.  N55                                 ....20.  $W$4T
.... 8.  6.9543E-5                           ....21.  78L7KUJ
.... 9.  /HJ2                                ....22.  /ID 838475,56
....10.  )                                   ....23.  42760.
....11.  447936460                           ....24.  =
....12.  .                                   ....25.  L6QIX
....13.  /FLIST

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.

          1.  A FORTRAN INTEGER CONSTANT
          2.  A FORTRAN INTEGER VARIABLE
          3.  A FORTRAN REAL CONSTANT
          4.  A FORTRAN REAL VARIABLE
          5.  NONE OF THE ABOVE

....26.  WY-                                 ....39.  2.70E+7
....27.  KS                                  ....40.  .449E-3
....28.  APVJK                               ....41.  .04639E+4
....29.  584                                 ....42.  447675023
....30.  *00590                              ....43.  J
....31.  .655147                             ....44.  EHHY$G5
....32.  .640176                             ....45.  JUPTAW47F
....33.  MN                                  ....46.  50.E+1
....34.  KOKLTP                              ....47.  725
....35.  PWK4Q(                              ....48.  3.E+3
....36.  5.46E-5                             ....49.  UYR
....37.  Y5Z                                 ....50.  U$QQR*S3447
....38.  37


 SCORING FORMULA = RIGHT*2             24.20
 MINIMUM SCORE = 10
```

Figure 1—Quiz 1 example



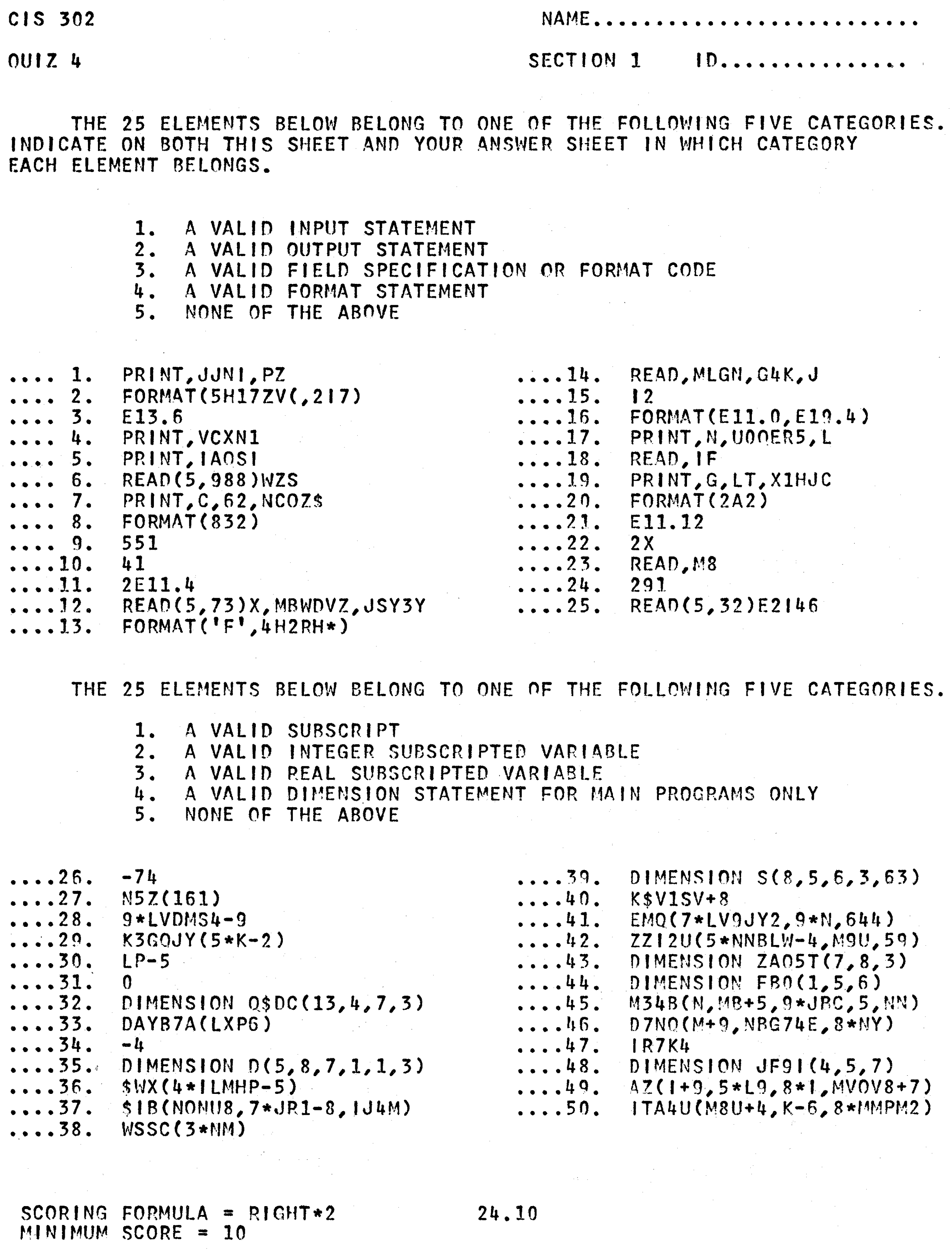
```
CIS 302                                          NAME.........................

QUIZ 1                                          SECTION 2     ID...............

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.
INDICATE ON BOTH THIS SHEET AND YOUR ANSWER SHEET IN WHICH CATEGORY
EACH ELEMENT BELONGS.

          1.  A FORTRAN IV SPECIAL CHARACTER
          2.  A FORTRAN IV CONSTANT
          3.  A FORTRAN IV SYMBOL
          4.  A VALID JOB CONTROL LANGUAGE COMMAND
          5.  NONE OF THE ABOVE

.... 1.  X$ZI=K                                 ....14.  I
.... 2.  =                                      ....15.  .78242E+9
.... 3.  .                                      ....16.  /V
.... 4.  .62522E+8                              ....17.  /LL4V7T
.... 5.  /FLIST                                 ....18.  CYOK6
.... 6.  9IW$0GM                                ....19.  ,
.... 7.  N                                      ....20.  /INSERT 9
.... 8.  51346984                               ....21.  4X/
.... 9.  PP6NK4LMV                              ....22.  1/7KG
....10.  OTE9AOK6                               ....23.  762059882
....11.  45048833                               ....24.  8.E+4
....12.  3.7                                    ....25.  KOV
....13.  /INTER

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.

          1.  A FORTRAN INTEGER CONSTANT
          2.  A FORTRAN INTEGER VARIABLE
          3.  A FORTRAN REAL CONSTANT
          4.  A FORTRAN REAL VARIABLE
          5.  NONE OF THE ABOVE

....26.  JJ8                                    ....39.  H3VOG
....27.  K2NP3                                  ....40.  5.74534E-3
....28.  PFR                                    ....41.  184
....29.  AZJVM7                                 ....42.  Q5401HOQUVT
....30.  41                                     ....43.  Y7OD+$7PO
....31.  H8Z                                    ....44.  9.04E+1
....32.  L3F$                                   ....45.  $8IOE4DL
....33.  SEEXOH                                 ....46.  HO3(
....34.  .86E+5                                 ....47.  8.28739E+4
....35.  VFKCY                                  ....48.  2
....36.  R*JVYP                                 ....49.  096
....37.  .6E-4                                  ....50.  143
....38.  9.E-2

  SCORING FORMULA = RIGHT*2              24.30
  MINIMUM SCORE = 10
```

Figure 2—Quiz 1 variation



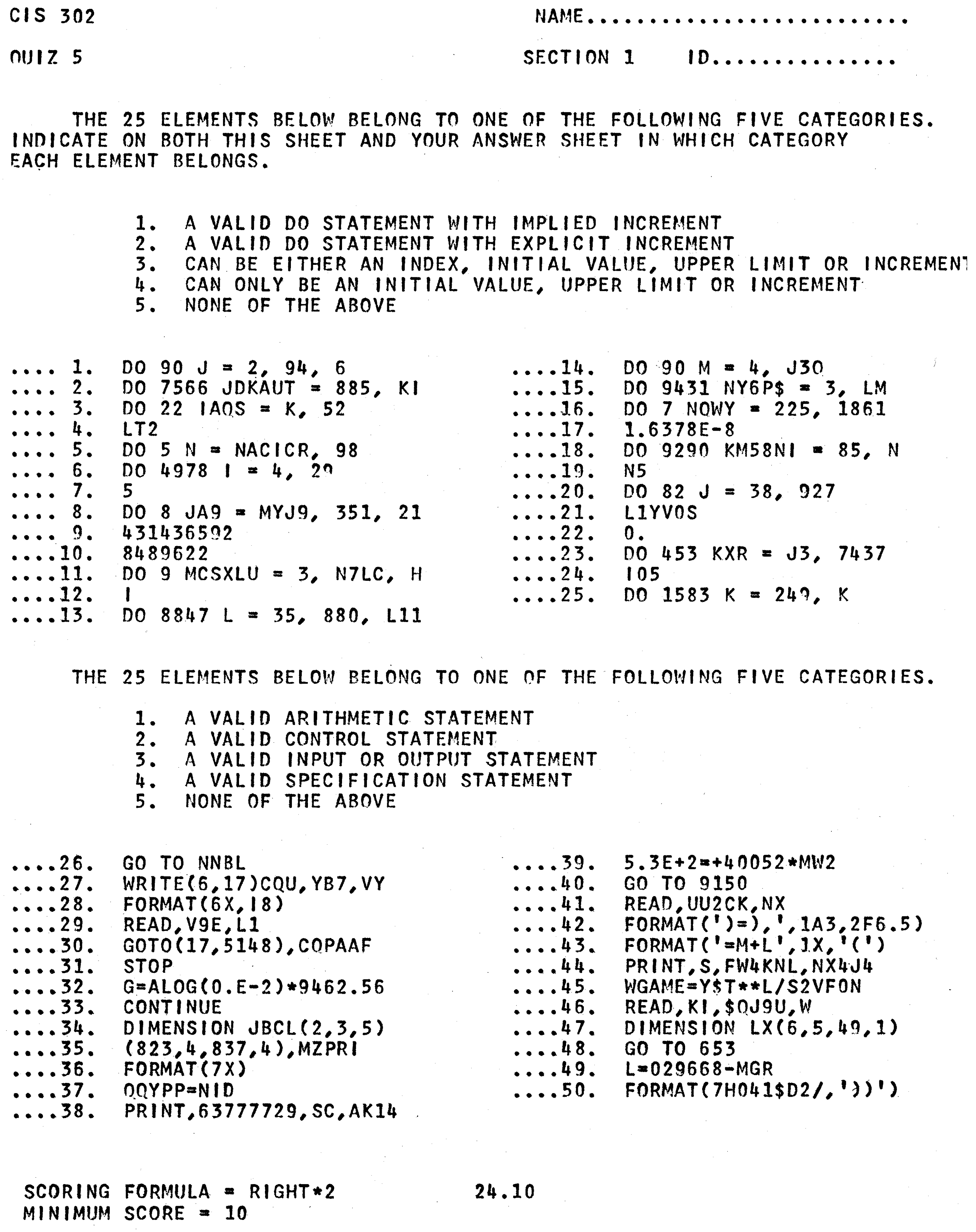
```
CIS 302                                           NAME........................

QUIZ 2                                          SECTION 1    ID...............

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.
INDICATE ON BOTH THIS SHEET AND YOUR ANSWER SHEET IN WHICH CATEGORY
EACH ELEMENT BELONGS.

          1.  AN EXPRESSION CONTAINING ONLY ONE MODE OF OPERAND (INTEGER OR RE
          2.  ALL OTHER EXPRESSIONS
          3.  A VALID ARITHMETIC STATEMENT
          4.  AN INVALID ARITHMETIC STATEMENT  (CONTAINS AN =SIGN)
          5.  NONE OF THE ABOVE

.... 1.  LGO9F9=(ITM-JSC)                     ....14.  ((7239+XDZU))
.... 2.  N55W=ALOG(.4/Z$D/.96)                ....15.  W=DROBLI)
.... 3.  28(                                  ....16.  Y1K)(L)
.... 4.  BI=NY7M-6                            ....17.  A,(395278364)
.... 5.  EXP((-0.56255))                      ....18.  S$X=((.99E-9))
.... 6.  -8+3                                 ....19.  2358(
.... 7.  (+(-L)                               ....20.  -W59UFX**1
.... 8.  K=(-JUPT21)+5                        ....21.  +JQ*1
.... 9.  COS(+M1J-D)                          ....22.  TO=*93296*9
....10.  ABS(COS(5.6960E+4**4))               ....23.  37=(LE)+9
....11.  9.48E+4=(-IX6RY)                     ....24.  +DE=65919
....12.  TANH(ZXH**JTHY)                      ....25.  .504111
....13.  ,((53)/L1S881)

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.

          1.  A STATEMENT CAUSING AN UNCONDITIONAL TRANSFER
          2.  A STATEMENT HAVING A 2 WAY CONDITIONAL TRANSFER   (ASSUME NO
          3.  A STATEMENT HAVING A 3 WAY CONDITIONAL TRANSFER   (ABNORMAL
          4.  A STATEMENT HAVING A 4 WAY CONDITIONAL TRANSFER   (TERMINATIONS
          5.  NONE OF THE ABOVE AND/OR A SYNTACTICALLY INCORRECT STATEMENT

....26.  GOTO(796,562,282),K18BWB             ....39.  GO TO 989
....27.  GO TO 175                            ....40.  (9,82,30,9525),IUK
....28.  GOTO(7886,65,9,1),INAIGO             ....41.  GOTO(514,65,648,8),K$J
....29.  GOTO(7,7,7),MYI                      ....42.  IF(-LGUZN)4,3,814
....30.  IF(NP-4)9,25,9                       ....43.  IF((RFG))16,4,22
....31.  GO TO 65                             ....44.  IF(QWO2/.5)5970,1,53
....32.  GOTO(77,5,402,5245),L81V             ....45.  GOTO(917,657,3433),I
....33.  GOTO(3),N3017                        ....46.  IF((DZTLY))2,2,2
....34.  GOTO(96,210,210,96),N                ....47.  GOTO(5,813,8,95),MOXO
....35.  GOTO(8,8,8,8),CFZ                    ....48.  GOTO(9,8383,8,48),NOIAO
....36.  IF((A))31,31,31                      ....49.  GOTO(4,1,2,5283),LRIGP9
....37.  (KL)3350,672,3350                    ....50.  IF(ALOG(W))376,413,413
....38.  GOTO(282,681,1,5),NKS


 SCORING FORMULA = RIGHT*2          24.20
 MINIMUM SCORE = 10
```

Figure 3—Quiz 2 example



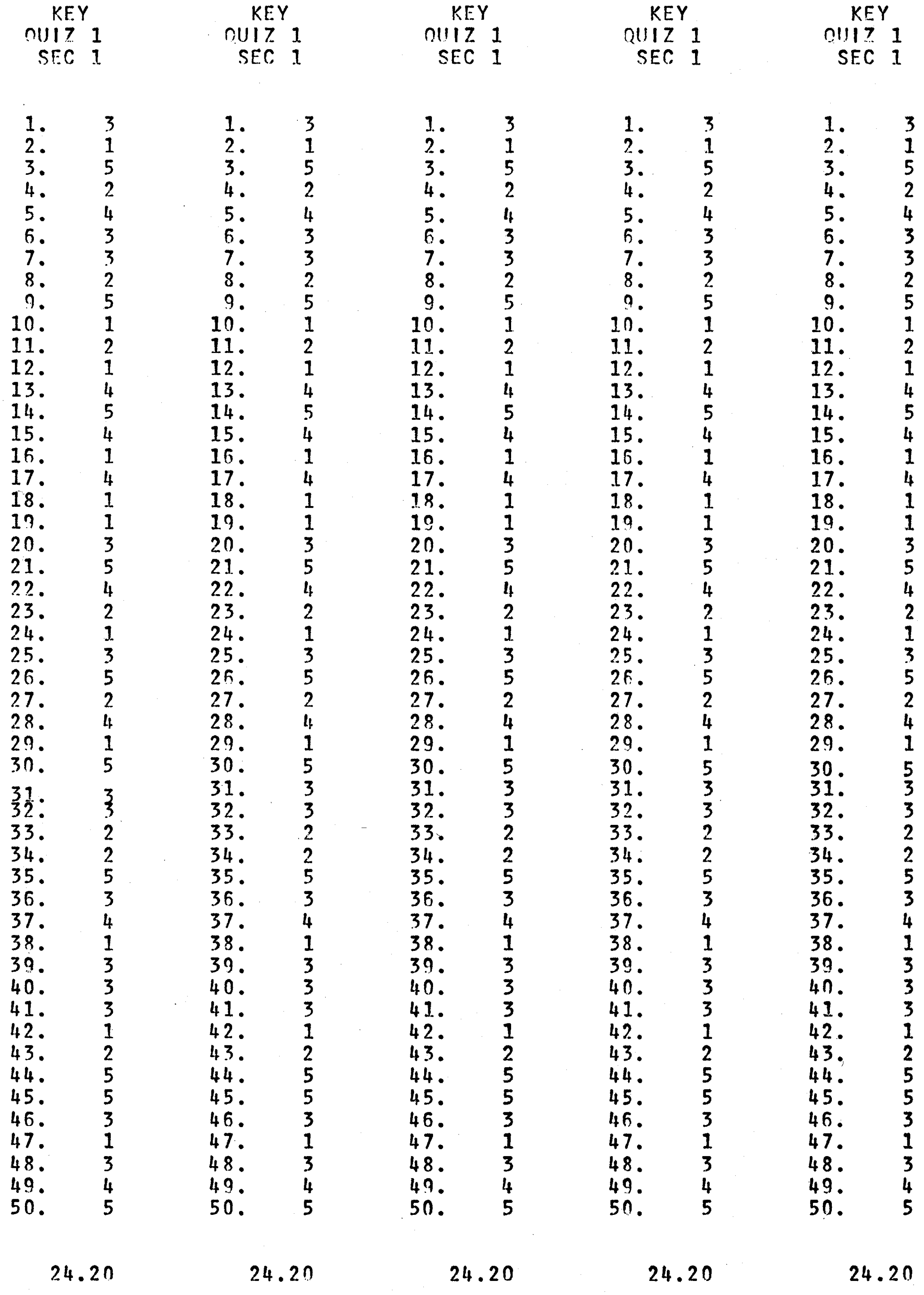
```
CIS 302                                      NAME.........................

QUIZ 4                                     SECTION 1    ID...............

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.
INDICATE ON BOTH THIS SHEET AND YOUR ANSWER SHEET IN WHICH CATEGORY
EACH ELEMENT BELONGS.

          1.  A VALID INPUT STATEMENT
          2.  A VALID OUTPUT STATEMENT
          3.  A VALID FIELD SPECIFICATION OR FORMAT CODE
          4.  A VALID FORMAT STATEMENT
          5.  NONE OF THE ABOVE

.... 1.  PRINT,JJNI,PZ                    ....14.  READ,MLGN,G4K,J
.... 2.  FORMAT(5H17ZV(,2I7)              ....15.  I2
.... 3.  E13.6                            ....16.  FORMAT(E11.0,E19.4)
.... 4.  PRINT,VCXN1                      ....17.  PRINT,N,UOOER5,L
.... 5.  PRINT,IAOSI                      ....18.  READ,IF
.... 6.  READ(5,988)WZS                   ....19.  PRINT,G,LT,X1HJC
.... 7.  PRINT,C,62,NCOZ$                 ....20.  FORMAT(2A2)
.... 8.  FORMAT(832)                      ....21.  E11.12
.... 9.  551                              ....22.  2X
....10.  41                               ....23.  READ,M8
....11.  2E11.4                           ....24.  291
....12.  READ(5,73)X,MBWDVZ,JSY3Y         ....25.  READ(5,32)E2I46
....13.  FORMAT('F',4H2RH*)

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.

          1.  A VALID SUBSCRIPT
          2.  A VALID INTEGER SUBSCRIPTED VARIABLE
          3.  A VALID REAL SUBSCRIPTED VARIABLE
          4.  A VALID DIMENSION STATEMENT FOR MAIN PROGRAMS ONLY
          5.  NONE OF THE ABOVE

....26.  -74                              ....39.  DIMENSION S(8,5,6,3,63)
....27.  N5Z(161)                         ....40.  K$V1SV+8
....28.  9*LVDMS4-9                       ....41.  EMQ(7*LV9JY2,9*N,644)
....29.  K3GOJY(5*K-2)                    ....42.  ZZI2U(5*NNBLW-4,M9U,59)
....30.  LP-5                             ....43.  DIMENSION ZA05T(7,8,3)
....31.  0                                ....44.  DIMENSION FB0(1,5,6)
....32.  DIMENSION Q$DC(13,4,7,3)         ....45.  M34B(N,MB+5,9*JBC,5,NN)
....33.  DAYB7A(LXP6)                     ....46.  D7NQ(M+9,NBG74E,8*NY)
....34.  -4                               ....47.  IR7K4
....35.  DIMENSION D(5,8,7,1,1,3)         ....48.  DIMENSION JF9I(4,5,7)
....36.  $WX(4*ILMHP-5)                   ....49.  AZ(I+9,5*L9,8*I,MVOV8+7)
....37.  $IB(NONU8,7*JR1-8,IJ4M)          ....50.  ITA4U(M8U+4,K-6,8*MMPM2)
....38.  WSSC(3*NM)


 SCORING FORMULA = RIGHT*2              24.10
 MINIMUM SCORE = 10
```

Figure 4—Quiz 4 example



```
CIS 302                                        NAME........................

QUIZ 5                                        SECTION 1    ID...............

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.
INDICATE ON BOTH THIS SHEET AND YOUR ANSWER SHEET IN WHICH CATEGORY
EACH ELEMENT BELONGS.

          1.  A VALID DO STATEMENT WITH IMPLIED INCREMENT
          2.  A VALID DO STATEMENT WITH EXPLICIT INCREMENT
          3.  CAN BE EITHER AN INDEX, INITIAL VALUE, UPPER LIMIT OR INCREMEN
          4.  CAN ONLY BE AN INITIAL VALUE, UPPER LIMIT OR INCREMENT
          5.  NONE OF THE ABOVE

.... 1.  DO 90 J = 2, 94, 6               ....14.  DO 90 M = 4, J30
.... 2.  DO 7566 JDKAUT = 885, KI         ....15.  DO 9431 NY6P$ = 3, LM
.... 3.  DO 22 IAQS = K, 52               ....16.  DO 7 NOWY = 225, 1861
.... 4.  LT2                              ....17.  1.6378E-8
.... 5.  DO 5 N = NACICR, 98              ....18.  DO 9290 KM58NI = 85, N
.... 6.  DO 4978 I = 4, 29                ....19.  N5
.... 7.  5                                ....20.  DO 82 J = 38, 927
.... 8.  DO 8 JA9 = MYJ9, 351, 21         ....21.  L1YV0S
.... 9.  431436592                        ....22.  0.
....10.  8489622                          ....23.  DO 453 KXR = J3, 7437
....11.  DO 9 MCSXLU = 3, N7LC, H         ....24.  I05
....12.  I                                ....25.  DO 1583 K = 249, K
....13.  DO 8847 L = 35, 880, L11

     THE 25 ELEMENTS BELOW BELONG TO ONE OF THE FOLLOWING FIVE CATEGORIES.

          1.  A VALID ARITHMETIC STATEMENT
          2.  A VALID CONTROL STATEMENT
          3.  A VALID INPUT OR OUTPUT STATEMENT
          4.  A VALID SPECIFICATION STATEMENT
          5.  NONE OF THE ABOVE

....26.  GO TO NNBL                       ....39.  5.3E+2=+40052*MW2
....27.  WRITE(6,17)CQU,YB7,VY            ....40.  GO TO 9150
....28.  FORMAT(6X,I8)                    ....41.  READ,UU2CK,NX
....29.  READ,V9E,L1                      ....42.  FORMAT(')=),',1A3,2F6.5)
....30.  GOTO(17,5148),COPAAF             ....43.  FORMAT('=M+L',1X,'(')
....31.  STOP                             ....44.  PRINT,S,FW4KNL,NX4J4
....32.  G=ALOG(0.E-2)*9462.56            ....45.  WGAME=Y$T**L/S2VF0N
....33.  CONTINUE                         ....46.  READ,KI,$QJ9U,W
....34.  DIMENSION JBCL(2,3,5)            ....47.  DIMENSION LX(6,5,49,1)
....35.  (823,4,837,4),MZPRI              ....48.  GO TO 653
....36.  FORMAT(7X)                       ....49.  L=029668-MGR
....37.  OQYPP=NID                        ....50.  FORMAT(7H041$D2/,'))')
....38.  PRINT,63777729,SC,AK14

  SCORING FORMULA = RIGHT*2             24.10
  MINIMUM SCORE = 10
```

Figure 5—Quiz 5 example



```
    KEY             KEY             KEY             KEY             KEY
  QUIZ 1          QUIZ 1          QUIZ 1          QUIZ 1          QUIZ 1
   SEC 1           SEC 1           SEC 1           SEC 1           SEC 1


  1.     3        1.     3        1.     3        1.     3        1.     3
  2.     1        2.     1        2.     1        2.     1        2.     1
  3.     5        3.     5        3.     5        3.     5        3.     5
  4.     2        4.     2        4.     2        4.     2        4.     2
  5.     4        5.     4        5.     4        5.     4        5.     4
  6.     3        6.     3        6.     3        6.     3        6.     3
  7.     3        7.     3        7.     3        7.     3        7.     3
  8.     2        8.     2        8.     2        8.     2        8.     2
  9.     5        9.     5        9.     5        9.     5        9.     5
 10.     1       10.     1       10.     1       10.     1       10.     1
 11.     2       11.     2       11.     2       11.     2       11.     2
 12.     1       12.     1       12.     1       12.     1       12.     1
 13.     4       13.     4       13.     4       13.     4       13.     4
 14.     5       14.     5       14.     5       14.     5       14.     5
 15.     4       15.     4       15.     4       15.     4       15.     4
 16.     1       16.     1       16.     1       16.     1       16.     1
 17.     4       17.     4       17.     4       17.     4       17.     4
 18.     1       18.     1       18.     1       18.     1       18.     1
 19.     1       19.     1       19.     1       19.     1       19.     1
 20.     3       20.     3       20.     3       20.     3       20.     3
 21.     5       21.     5       21.     5       21.     5       21.     5
 22.     4       22.     4       22.     4       22.     4       22.     4
 23.     2       23.     2       23.     2       23.     2       23.     2
 24.     1       24.     1       24.     1       24.     1       24.     1
 25.     3       25.     3       25.     3       25.     3       25.     3
 26.     5       26.     5       26.     5       26.     5       26.     5
 27.     2       27.     2       27.     2       27.     2       27.     2
 28.     4       28.     4       28.     4       28.     4       28.     4
 29.     1       29.     1       29.     1       29.     1       29.     1
 30.     5       30.     5       30.     5       30.     5       30.     5
 31.     3       31.     3       31.     3       31.     3       31.     3
 32.     3       32.     3       32.     3       32.     3       32.     3
 33.     2       33.     2       33.     2       33.     2       33.     2
 34.     2       34.     2       34.     2       34.     2       34.     2
 35.     5       35.     5       35.     5       35.     5       35.     5
 36.     3       36.     3       36.     3       36.     3       36.     3
 37.     4       37.     4       37.     4       37.     4       37.     4
 38.     1       38.     1       38.     1       38.     1       38.     1
 39.     3       39.     3       39.     3       39.     3       39.     3
 40.     3       40.     3       40.     3       40.     3       40.     3
 41.     3       41.     3       41.     3       41.     3       41.     3
 42.     1       42.     1       42.     1       42.     1       42.     1
 43.     2       43.     2       43.     2       43.     2       43.     2
 44.     5       44.     5       44.     5       44.     5       44.     5
 45.     5       45.     5       45.     5       45.     5       45.     5
 46.     3       46.     3       46.     3       46.     3       46.     3
 47.     1       47.     1       47.     1       47.     1       47.     1
 48.     3       48.     3       48.     3       48.     3       48.     3
 49.     4       49.     4       49.     4       49.     4       49.     4
 50.     5       50.     5       50.     5       50.     5       50.     5

    24.20           24.20           24.20           24.20           24.20
```

Figure 6—Key example

the first random number in this loop would be limited to a value between 1 and 6. Subsequent random numbers produced in this loop would be between 1 and 37 corresponding to the 26 letters, 10 digits and the $ sign. Thus, for Question 27, the characters KS resulted. In similar fashion, the names for Questions 33, 34, and 43 were produced.

As each category for each question is determined by the main program, the values between 1 and 5 are kept in a table to be used as the answer key. This table is listed for each quiz and section as shown in Figure 6 for use in class after quiz administration is complete. A card is also punched containing the key for input to a computerized grading system which is used to grade tests and homework and maintain records for the course.

To illustrate the scope of this quiz generator in terms of programming effort, the following list gives the name and purpose of each subroutine in the total package. Each routine is written in Fortran IV:

Name	*Purpose*
MAIN	General test formatting and key production
SETUP	Prints a leader to help operator setup printer
QUIZi	Calls routines for categories in each quiz
ALPNUM	Generates single alphanumeric characters
SYMBOL	″ a Fortran symbol
CONSTA	″ ″ ″ constant, real or integer
SPECHA	″ ″ ″ special character
JCLCOM	″ ″ job command
NONEi	″ none of the above entries for each quiz
INTCON	″ a Fortran integer constant
INTEXP	″ ″ ″ ″ expression
REAEXP	″ ″ ″ real ″
MIXEXP	″ ″ ″ mixed ″
MIXILE	″ ″ illegal expression
UNIARY	″ ″ uniary operator expression
PAREN	″ ″ expression within parentheses
BINARY	″ ″ binary operator expression
FUNCT	″ ″ function call
ARITH	″ ″ Fortran arithmetic statement
GOTON	″ ″ ″ GO TO statement
IFSTAT	″ ″ ″ IF ″
COGOTO	″ ″ ″ comp GO TO ″
INOUT	″ ″ ″ I/O statement
FIESPE	″ ″ format field specification
FORMAT	″ ″ format statement
DOSTAT	″ ″ Fortran DO statement
SIZCON	″ ″ constant of a given size
CONTRL	″ ″ control statement
SPESTA	″ ″ specification statement
INTVAR	″ ″ integer variable
REACON	″ ″ real constant
REAVAR	″ ″ ″ variable
STANUM	″ ″ statement number
SUBSCR	″ ″ subscripted variable
INTSUB	″ ″ integer ″ ″
REASUB	″ ″ real ″ ″
DIMENS	″ ″ dimension statement

The only major criticism that can be made of these quizzes is that they fail to test the student on his understanding of the behavior of the computer under the control of these various statements either singly or in combination. This understanding of the semantics of Fortran, of course, is imperative if a programmer is to be successful. Thus a method is needed for generating questions which will test the student in this understanding. It is this problem the solution of which is now being sought. The following sections describe some of the major concepts discovered so far and possible methods of solution.

SEMANTIC QUESTION GENERATION

Work is now under way for designing a system to produce questions which require semantic understanding as well as syntactic recognition of various Fortran program segments. The major difficulties in such a process is the determination of the correct answer for the generation of a key and the computation of the most probable incorrect answers for the distractors of a question. Both of these determinations sometimes involve semantic meanings (i.e., evaluation of expressions or the execution of statements) which would be difficult to determine in the same program that generates the question element in the first place. As a good illustration, consider the following question model:

Given the following statement:

IF (X + 2.0 – SQRT(A)) 5,27,13
where X = 6.5
and A = 22.7
Transfer is made to the statement whose number is
(1) 5 (2) 27 (3) 13 (4) indeterminant
(5) none of the above as
the statement is invalid

Here the generator would have created the expression X + 2.0 – SQRT(A), the three statement numbers 5, 27 and 13 and finally the two values of X and A. The order of the first four answer choices could also be determined randomly. In this particular question, determination of the distractors is no problem but the determination of the correct answer involves an algorithm similar to the following:

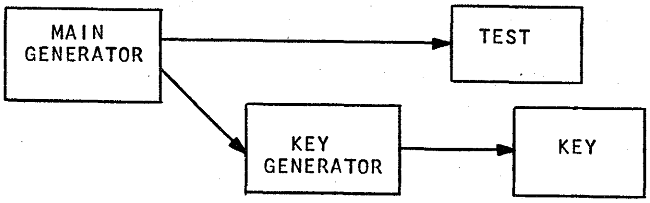


Figure 7—2nd stage involvement of key

```
     X = 6.5
     A = 22.7
     IF (X + 2.0 – SQRT(A)) 5, 27, 13
 5   KEY = 1
     GO TO 10
27   KEY = 2
     GO TO 10
13   KEY = 3
10   CONTINUE
```

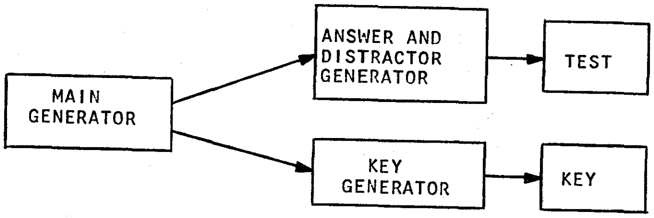


Figure 8—2nd stage involvement of key and distractors

This problem can be solved by letting the main generator program generate a second program to compute the key as well as generate the question for the test. This second program would then be passed to further job steps which would compile and execute the program and determine the key for the question. Figure 7 illustrates this concept.

As an illustration of a question involving more difficult determination of answer and distractors, the following question model is presented.

Given the statement:

$$I = J/2 + X$$

$$\text{where } J = 11$$

$$\text{and } X = 6.5$$

the resulting value of I is

(1) 11.5 (2) 11 (3) 12 (4) 6.5 (5) 6

The determination of the five answer choices would have to be determined by an algorithm such as the

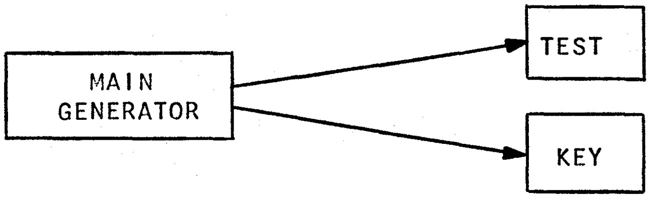
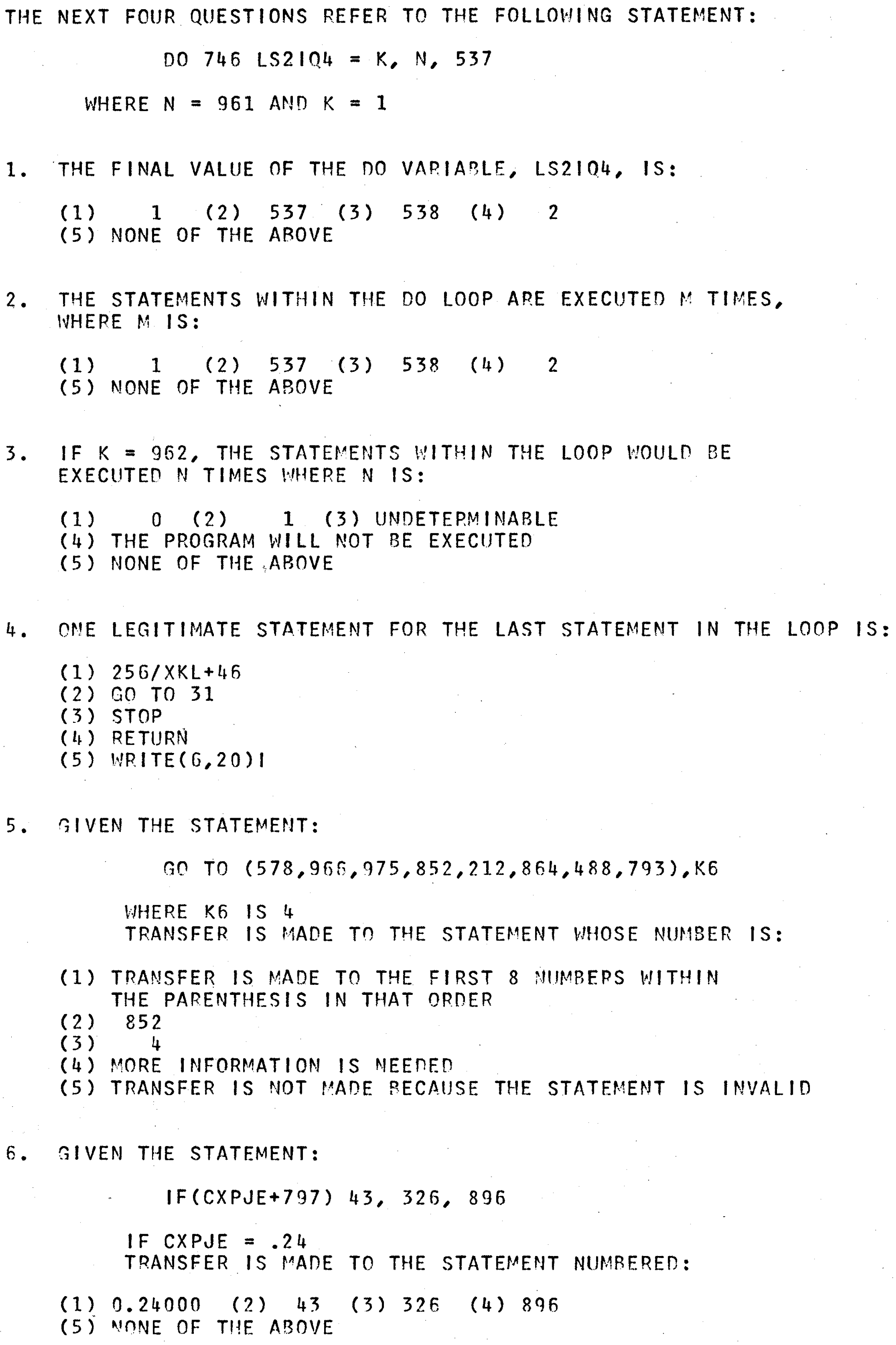


Figure 9—No 2nd stage involvement



```
THE NEXT FOUR QUESTIONS REFER TO THE FOLLOWING STATEMENT:

              DO 746 LS2IQ4 = K, N, 537

       WHERE N = 961 AND K = 1


1.   THE FINAL VALUE OF THE DO VARIABLE, LS2IQ4, IS:

     (1)    1   (2)  537  (3)  538  (4)   2
     (5) NONE OF THE ABOVE


2.   THE STATEMENTS WITHIN THE DO LOOP ARE EXECUTED M TIMES,
     WHERE M IS:

     (1)    1   (2)  537  (3)  538  (4)   2
     (5) NONE OF THE ABOVE


3.   IF K = 962, THE STATEMENTS WITHIN THE LOOP WOULD BE
     EXECUTED N TIMES WHERE N IS:

     (1)    0  (2)    1  (3) UNDETERMINABLE
     (4) THE PROGRAM WILL NOT BE EXECUTED
     (5) NONE OF THE ABOVE


4.   ONE LEGITIMATE STATEMENT FOR THE LAST STATEMENT IN THE LOOP IS:

     (1) 256/XKL+46
     (2) GO TO 31
     (3) STOP
     (4) RETURN
     (5) WRITE(6,20)I


5.   GIVEN THE STATEMENT:

              GO TO (578,966,975,852,212,864,488,793),K6

          WHERE K6 IS 4
          TRANSFER IS MADE TO THE STATEMENT WHOSE NUMBER IS:

     (1) TRANSFER IS MADE TO THE FIRST 8 NUMBERS WITHIN
         THE PARENTHESIS IN THAT ORDER
     (2)  852
     (3)    4
     (4) MORE INFORMATION IS NEEDED
     (5) TRANSFER IS NOT MADE BECAUSE THE STATEMENT IS INVALID


6.   GIVEN THE STATEMENT:

              IF(CXPJE+797) 43, 326, 896

          IF CXPJE = .24
          TRANSFER IS MADE TO THE STATEMENT NUMBERED:

     (1) 0.24000  (2)   43  (3) 326  (4) 896
     (5) NONE OF THE ABOVE
```

Figure 10—Semantic question examples

following:

$$J = 11$$

$$X = 6.5$$

$$\text{ANS1} = J/2 + X$$

$$\text{IANS2} = J/2 + X$$

$$\text{IANS3} = J/2. + X$$

$$\text{ANS4} = X$$

$$\text{IANS5} = X$$

In this problem not only does the determination of the key depend on further computation but also the distractors and the correct answer. Thus the second program generated by the first program must be involved in the production of the test as well as the key. Figure 8 illustrates this concept.

Some questions are very simple to produce as neither key nor answer choices depend on a generated algorithm. An example is:

Given the following statement:

DO 35 $J5 = 3, 28, 2$

The DO loop would normally be iterated N times where N is

(1) 13 (2) 12 (3) 14 (4) 28 (5) 35

Here the answer choices are determined from known algorithms independent of the random question elements. No additional program is therefore required for producing this test question and its key. Figure 9 illustrates this condition.

It would then appear that a general semantic test generator would have to satisfy at least the conditions exhibited in Figures 7, 8 and 9.

Figure 10 illustrates results obtained from a working pilot program utilizing the method illustrated in Figure 8. This program is a very complicated one and was very difficult to write. To produce a Fortran program as output from a Fortran program involved a good deal of tedious work such as writing Format statements within Format statements. It has become obvious that a more reasonable method of writing the source program is needed.

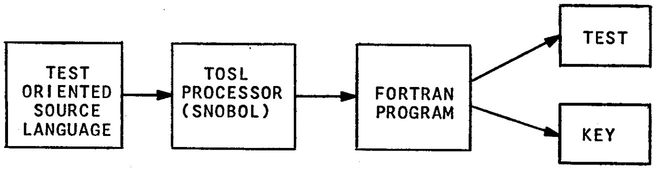


Figure 11—TOSL Language environment

FUTURE INVESTIGATION

An attempt will be made to design a source language oriented toward test design which will then be translated by a new processor into a Fortran program. See Figure 11.

This new language is visualized as being composed of a mixture of languages including the possibility of passing simple English statements (for the textural part of a question) through the entire process to the test. Fortran statements could be written into the source language where such algorithms are required. Finally, statements to allow the specification of random question elements and the linkage of these random elements to the algorithms mentioned above will be necessary.

Several special source language operators can be introduced to facilitate the writing of question models. Certain special characters can be chosen to represent particular requirements such as question number control, random variable control, answer choice control, answer choice randomization, and key production. It is anticipated that SNOBOL would make an excellent choice for the processor language as it will allow for rapid recognition of the source language elements and operations and in a natural way generate and maintain strings which will find their way into the Fortran output program and finally into the test and key. The possibilities of such a system look very promising and hopefully, such a system can be made applicable to other subject fields as well as the current one.

A conversational item banking and test construction system

by FRANK B. BAKER

University of Wisconsin
Madison, Wisconsin

INTRODUCTION

Most conscientious college instructors maintain a pool of items to facilitate the construction of course examinations. Typically, each item is typed on a 5″×8″ card and coded by course, book chapter, concept and other such keys. The back of the card usually contains data about the item collected from one or more administrations of the item. To construct a test, the instructor peruses this item bank looking for items that meet his current needs. Items are selected on the basis of their content and further filtered by examining the item data on the card, overlapping items are eliminated, and the emphasis of the test is balanced. After having maintained such a system for a number of years, it became obvious that there should be a better way. Consequently, the total process of maintaining an item bank and creating a test was examined in detail. The result of this study was the design and implementation of the Test Construction and Analysis Program (TCAP). The design goal was to provide an instructor with a computer based item banking and test construction system. Because the typical instructor maintains a rather modest item bank, the design emphasis was upon flexibility and capabilities rather than upon capacity. In order to achieve the necessary flexibility TCAP was implemented as a conversational system using an interactive terminal. Considerable care was taken to build a system that had a very simple computer-user interface.

The purpose of the present paper is to describe the TCAP system. The order of discussion proceeds from the file structure to the software to the use of the system. This particular order enables the reader to see the underlying system logic without becoming enmeshed in excessive interaction between components.

SYSTEM DESIGN

File structure

The three basic files of the TCAP system are the Item, Statistics and Test files. A record in the Item file contains the actual item and is a direct analogy to the 5″×8″ card of the manual scheme. A record in the Statistics file contains item analysis results for up to ten administrations of a given item. Test file records contain summary statistics for each test that has been administered. The general structure of all files is essentially the same although they vary in internal detail. Each file is preceded by a header (see Figure 1) that describes the layout of the record in the file. Because changing computers has been a way of life for the past ten years, the header specifies the number of bits per character and number of characters per word of the target computer. These parameters are used to make the files word length independent. In addition, it contains the number of sections per record, the number of characters per record section, characters per record and the number of records in the file. The contents of the headers allow all entries to data items within a record to be located via a relative addressing scheme based upon character counts. This character oriented header scheme enables one to arbitrarily specify the record size and layout at run time rather than compile time; thus, enabling several different users of the system to employ their own record layouts without affecting the TCAP software.

A record is divided into sections of arbitrary length, each preceded by a unique two character flag and terminated by a double period. Sub sections within a section are separated by double commas. These flags serve a number of different functions during the file creation phase and facilitate the relative addressing scheme used to search within a record. Figure 2 contains an item

File Header Element	Contents
1	Name of file
2	Number of bits per character in target computer
3	Characters per word in the target computer
4	Characters per record in the file
5	Number of sections in the record
6-15	Number of characters in section i where $i = 1,2,\ldots 10$

Figure 1—Typical file header

file record that represents a typical record layout. The basic record layout scheme is the same in all files, but they differ in the contents of the sections. A record in the item file consists of seven sections: Identification, Keyword, Item, Current item statistics, Date last used, and Frequency of use, previous version identification. The ID section contains a unique identification code for the item that must begin with *$. The keyword section contains free field keyword descriptors of the item separated by commas. The item section contains the actual item and was intended primarily for multiple choice items. Since the item section is free field, other item types could be stored, but it has not been tried to date. The current item statistics section stores the item analysis information from the most recent administration of the item. The first element of this section is the identification code of the test from which the item statistics were obtained. The internal layout of this section is fixed so that the FORTAP item analysis program outputs can be used to update the information. The item statistics section contains information such as the number of persons selecting each item response, item difficulty, and estimates of the item parameters. The next section contains the date of the most recent administration of the item. The following section contains a count of the total number of times the item has been administered. These two pieces of information are used in the test construction section to prevent over use of an item. The final section of the item record contains the unique identification code of a previous version of the same item. This link enables one to follow the development of a given item over a number of modifications.

A record in the Statistics file contains 11 sections, an item identification section and 10 item statistics sections identical in format to the current item statistics section of the item record. These 10 sections are maintained as a first in, last out push down stack with an eleventh data set causing the first set to be pushed end off. Records in the Test file are similar to those of the Item file and have five sections: Identification, Keywords, Comments, Summary statistics of the test, and a link to other administrations of the same test. The comments section allows the instructor to store any anecdotal information he desires in a free field format. The link permits keeping track of multiple uses of the same test such as occurs when a course has many sections.

The record layouts were designed so that there was a one to one correspondence between each 72 characters in a section and the punched cards used to create the file. Such a correspondence greatly facilitates the ease with which an instructor can learn to use the system. Once he has key punched his item pool, the record layouts within each file are quite familiar to him and the operations upon these records are easily understood. This approach also permitted integration of the FORTAP item analysis program into the TCAP system with a minimum conversion effort.

It should be noted that the file design allows many different instructors to keep their items in the same basic files. Alternatively, each instructor can maintain

Item File Record

```
*$ STAT 01 520170. .
ZZ EDPSY,STATISTICS,ESTIMATORS,MLE. .
QQ ONE OF THE CHARACTERISTICS OF MAXIMUM LIKELIHOOD ESTIMATORS IS THAT IF SUFFICIENT ESTI-
MATES EXIST, THEY WILL BE MAXIMUM LIKELIHOOD ESTIMATORS. ESTIMATES ARE CONSIDERED SUFFI-
CIENT IF THEY, ,
(A) USE ALL OF THE DATA IN THE SAMPLE, ,
(B) DO NOT REQUIRE KNOWLEDGE OF THE POPULATION VALUE, ,
(C) APPROACH THE POPULATION VALUE AS SAMPLE SIZE INCREASES, ,
(D) ARE NORMALLY DISTRIBUTED.
WW TEST 01 220170. .
1 1 0 0014 .18 − .21 −01.36 −0.22, ,
1 2 1 0054 .69 + .53 −00.93   .63, ,
1 3 0 0010 .12 + .64 −01.77 −0.83. .
VV 161271. .
YY 006. .
$$ STAT 02 230270. .
```

Figure 2—A record in the item file

his own unique set of basic files, yet, use a common copy of the TCAP program. The latter scheme is preferred as it minimizes file search times.

Software design

The basic programming philosophy adopted was one of cascaded drivers with several levels of utility routines. Such an approach enables the decision making at each functional level to be controlled by the user interactively from a terminal. It also enables each level of software to share lower level utility routines appropriate to its tasks. Figure 3 presents a block diagram of the major software components of the TCAP system. The main TCAP driver is a small program that merely presents a list of operational modes to the user: Explore, Construct, and File Maintenance. Selection of a particular mode releases control to the corresponding next lower level driver. These second level drivers have access to four search routines that form a set of high level utility routines. The Identification search routine enables one to locate a record in a file by its unique identification code. The Keyword search routine implements a search of either the item or test file for records containing the combination of keywords specified by the user. At present a simple conjunctive match is used, but more complex logic can be added easily. The Parameter search utility searches the item or statistics files for items whose item parameter values fall within bounds specified by the user. The Linked search routine allows one to link from a record in one file to a corresponding record in another file. For example, from the item file to the statistics file or from the item file to the test file. Due to the extremely flexible manner in which the user can interact with the three files it was necessary to access these four search routines through the Basic File Handling routine. The BFH routine initializes the file handlers from the parameters in the headers, coordinates the file pointers, and handles certain error conditions. Such centralization relieves both the mode implementation routines and the search routines of considerable internal bookkeeping related to file usage. The four search routines in turn have access to a lower level of utility routines, not depicted in Figure 3. These lowest level utilities are routines that read and write records, pack and unpack character strings, convert numbers from alphanumeric to integer or floating point, and handle communication with the interactive terminal.

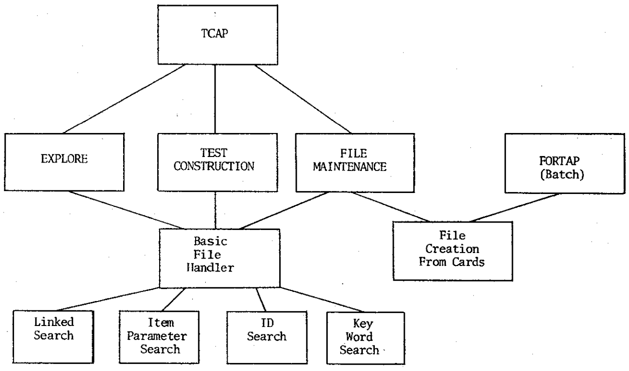


Figure 3—TCAP software structure

The purpose of the EXPLORE routine is to permit the user to peruse the three basic files in a manner analogous to thumbing through a card index. The EXPLORE routine presents the user with a display listing seven functions related to accessing records within a file. These functions are labeled: Identification, Keyword, Parameter, Linked, Restore, Mode and Continue. The first four of these correspond to the four utility search routines. The Restore option merely reverses the linkage process and causes the predecessor record to become the active record. The Mode option causes an exit from the EXPLORE routine and a return to the Mode display of the TCAP driver. The Continue option allows one to continue a given search using the present set of search specifications.

The Test Construction Routine is used to assemble an educational test from the items in the item file. Test construction is achieved by specifying a set of general characteristics all items should have and then defining sub sections of the test called areas. The areas within the test are defined by user supplied keywords and the number of items desired in an area. The Test Construction routine then employs the Keyword search routine, via BFH, to locate items possessing the proper keywords. This process is continued until the specified number of items for an area are retrived or the end of the item file is reached. Once the requirements of an area are satisfied the user is free to define another area or terminate this phase. Upon termination certain summary data, predicted test statistics, and the items are printed.

The function display of the File Maintenance routine presents the user with three options: Create, FORTAP and Single. The Create option is a batch mode process that uses the File Creation from Cards subroutine (FCC) to create any of the three basic files from a card deck. To use this option, it is necessary to simulate, via cards, the interaction leading to this point. The FORTAP option is interactive, but it assumes that the FORTAP item analysis routine has created a card image drum file containing the test and item analysis results. The file contains the current item statistics section for each item in the test accompanied by the appropriate identification sections and test links. A test file record for the test is also in this file. The File Maintenance routine transfers the current item statistics section of the item record of each item in the test to the corresponding record in the statistics file. It then uses the FCC subroutine to replace the current item statistics section of the item records with the item statistics section from the FORTAP generated file. If an item record does not exist in the Item file a record is created containing only the identification sections and the current item statistics. The test record is then stored in the Test file and the header updated. The Single option is used to perform line item updates on a single file. Under this option the File Maintenance routine assumes that card images are stored in an update file and that only parts of a given record are to be changed.

OPERATION OF THE SYSTEM

The preceding sections have described the file structure and the software design. The present section describes some interactive sequences representing typical uses of the TCAP system. The sequences contained in Figure 4 have had the lengthy record printouts deleted. The paragraphs below follow these scripts and are intended to provide the reader with a "feel" for the system operation.

Upon completion of the usual remote terminal sign in procedures, the TCAP program is entered and the mode selection message—TYPE IN TCAP MODE= EXPLORE, CONSTRUCT, FILE MAINTENANCE is printed at the terminal. The user selects the appropriate mode, say EXPLORE, by typing the name. The computer replies by printing the function display message. In the EXPLORE mode, this message is the list of possible search functions. The user responds ty typing the name of the function he desires to perform, keyword in the example. The computer responds by asking the user for the name of the file he wishes to search. Next, the user is instructed to type in the keywords separated by commas and terminated by a double period. The user must be aware of the keywords employed to describe the items and tests in the files. Hence, it is necessary to maintain a keyword dictionary external to the system. This should cause little trouble as the person who created the files is also the person using the system. Upon receipt of the keywords, the EXPLORE routine calls the Keyword Search routine to find an item containing the particular set of keywords. The contents of the item record located are then typed at the terminal. At this point the system asks the user for further instructions. It presents the message FUNCTION DISPLAY NEEDED. A negative reply causes a return to the Mode selection display of the TCAP driver. A YES response causes the EXPLORE function list to reappear. If one wishes to find the next item in the file possessing the same keyword pattern, CONTINUE, is typed and the search proceeds from the last item found. In Figure 4 this option was not selected. Returning to the Mode selection or reaching the end of the file being searched causes the Basic File Handler to restore the file pointers to the file origin.

The next sequence of interactions in Figure 4 links from a record in the Item file to the corresponding record in the Statistics file. It is assumed that one of the other search functions has been used to locate a record prior to selection of the LINKED option, the last item found via the Keyword search in the present example. The computer then prompts the user by asking for the name of the file from which the linking takes place, item in the present example. It then asks for the name of the file the user wishes to link to statistics in the example. There are several illegal linkages and the Linked search routine checks for a legal link. The Linked search routine extracts the identification section of the item record and establishes the inputs to the Identification Search routine. This routine then searches the Statistics file for a record having the same identification section. It should be noted that a utility routine used a utility routine at this point, but the cascaded control was supervised by the EXPLORE routine. When the proper Statistics record is found its contents are printed at the terminal. Again, the system asks for directions and the user is asked if he desires the function display. In the example, the user obtained the function display and selected the Restore option. This results in the prior record, the item record, being returned to active record status and the name of the active file being printed. The system allows one to link and restore to a depth of three records. Although not shown in the example sequences, the other options under the EXPLORE mode operate in an analogous fashion.

The third sequence of interactions in Figure 4 shows the construction of an examination via the TCAP system. Upon selection of the Construct mode, the computer instructs the user to supply the general item specifications, namely the correct response weight and the bounds for the item parameters X_{50} and β. These minimum, maximum values are used to filter out items having poor statistical properties. The remainder of the test construction process consists of using keywords to define areas within the test. The computer prints AREA DEFINITION FOLLOWS: YES, NO. After receiving a YES response the computer asks for the number of items to be included in the area. The user can specify any reasonable number, usually between 5 and 20. The program then enters the normal keyword search

```
  TYPE IN TCAP MØDE=EXPLØRE, CØNSTRUCTIØN, FILE MAINTENCE EXPLØRE
    FUNCTIØN DISPLAY
  TYPE KIND ØF SEARCH DESIRED
  IDENT,KEYWØRD,PARAMETER,LINKED,RESTØRE,CØNTINUE,MØDE
KEYWORD
    TYPE IN FILE NAME
ITEM
  TYPE IN KEYWØRDS SEPARATED BY CØMMAS
TERMINATE WITH . .
SKEWNESS,MEAN,MEDIAN. .
 *$AAAC 02 230270. .
      THE ITEM RECØRD WILL BE PRINTED HERE

FUNCTIØN DISPLAY NEEDED YES,NØ
YES
      FUNCTIØN DISPLAY
    TYPE KIND ØF SEARCH DESIRED
  IDENT,KEYWØRD,PARAMETER,LINKED,RESTØRE,CØNTINUE,MØDE
LINKED
  LINKED SEARCH REQUESTED
  TYPE NAME ØF FILE FRØM
ITEM
  TYPE NAME ØF FILE LINKED TØ
STAT
 *$AAAC 02 230270. .
        THE STATISTICS RECØRED WILL BE PRINTED HERE

FUNCTIØN DISPLAY NEEDED YES,NØ
YES
      FUNCTIØN DISPLAY
    TYPE KIND ØF SEARCH DESIRED
  IEDNT,KEYWØRD,PARAMETER,LINKED.RESTØRE,CØNTINUE,MØDE
RESTØRE
  ITEM RECØRD FILE RESTØRED
FUNCTIØN DISPLAY NEEDED YES,NØ
YES
      FUNCTIØN DISPLAY
  IDENT,KEYWØRD,PARAMETER,LINKED,RESTØRE,CØNTINUE,MØDE
MØDE
  TYPE IN TCAP MØDE=EXPLØRE,CØNSTRUCTIØN,FILE MAINTENANCE
CØNSTRUCT
  TYPE IN WEIGHT ASSIGNED TØ ITEM RESPØNSE
1
  TYPE IN MINIMUM VALUE ØF ×50
−2.5
  TYPE IN MAXIMUM VALUE ØF ×50
+2.5
  TYPE IN MINIMUM VALUE ØF BETA
.20
  TYPE IN MAXIMUM VALUE ØF BETA
1.5
  AREA DEFINITIØN FØLLØWS YES,NØ
YES
  TYPE IN NUMBER ØF ITEMS NEEDED FØR AREA
10
  TYPE IN KEYWØRDS SEPARATED BY CØMMAS
  TERMINATE WITH . .
CHAPTER1,STATISTICS,THEØRY,FISHER. .
  AREA DEFINITIØN FØLLØWS YES,NØ
YES
  TYPE IN NUMBERS ØF ITEMS NEEDED FØR AREA
10
```

Figure 4—Operational sequences

```
  TYPE IN KEYWØRDS SEPARATED BY CØMMAS
  TERMINATE WITH . .
CHAPTER2,DISTRIBUTIØN,FREQUENCY,INTERVAL. .
  AREA DEFINITIØN FØLLØWS YES,NØ
YES
  TYPE IN NUMBER ØF ITEMS NEEDED FØR AREA
10
  TYPE IN KEYWØRDS SEPARATED BY CØMMAS
  TERMINATE WITH . .
CHAPTER3,BINØMIAL,PARAMETER,CØMBINATIØN,PERMUTATIØN. .
  AREA DEFINITIØN FØLLØWS YES,NØ
YES
TYPE IN NUMBER ØF ITEMS NEEDED FØR AREA
10
  TYPE IN KEYWØRDS SEPARATED BY CØMMAS
  TERMINATE WITH . .
CHAPTER4,HYPØTHESES,LARGE SAMPLE,Z TEST. .
  AREA DEFINITIØN FØLLØWS YES,NØ
NØ
  ITEMS REQUESTED PER AREA    10  10  10  10
  ITEMS FØUND PER AREA         6   9   8  10
  PREDICTED TEST STATISTICS
MEAN=  16.0758
STANDARD DEVIATIØN=  4.561111
RELIABILITY=  .893706
  DØ YØU WANT ITEMS PRINTED YES,NØ
NØ
      ITEM IDENTIFICATIØN        X50                BETA
         1*$AAAA 03 230270. .              .470000              .450000
              (THIS INFØRMATIØN WILL BE PRINTED FØR ALL ITEMS)
  TYPE IN TCAP MØDE=EXPLØRE,CØNSTRUCTIØN,FILE MAINTENANCE
EXIT
      THAT IS END ØF RUN,GØØDBY
```

Figure 4—(Continued)

procedures and the user enters the keywords that define this area of the test. Upon receipt of the keywords the item file is searched for items possessing the proper descriptors and whose item parameters are within bounds. Completion of the keyword search results in a return to the area definition message. The area definition and search process can be repeated up to ten times. A NO response to the area definition message results in the printing of the table showing the number of items requested per area and the number actually found per area. The table is followed by the predicted values of the test mean, standard deviation, and internal consistency reliability index. These values are computed from the current values of the item parameters X_{50} and β of the retrieved items. These predicted values assist the test constructor in determining if an appropriate set of items has been selected by the system. The program then asks the user if he wants the selected items printed. If not, only the identification section and the values of the item parameters are printed. This information allows one to use the Identification search option of the EXPLORE routine to retrieve the items at a later date. A minor deficiency of the present test construction procedures is that a reproducible copy of the test is not produced. A secretary uses the hard copy to prepare a stencil or similar master. With some minor programming this final step could be accomplished.

Some enhancements

At the present time the full TCAP design has not been implemented and a number of additional features should be mentioned. Two sections of the item record, date of use, and frequency of use can be employed to prevent over use of the same items. A step in the test construction mode will enable the user to specify that an item used since a certain date or more than a specified number of times should not be retrieved. The software for this additional filtering has been written but not debugged.

A significant enhancement is one that enables the test constructor to manipulate the items constituting a test. For example, an instructor may not be satisfied with the items the computer has retrieved in certain areas. He may wish to delete items from one area and add items to another. This can be done interactively and the predicted test statistics should be re-calculated as each transaction occurs. At the present time, such manipulations require a re-run of the total test construction process. An extension allowing considerable freedom in manipulating items of the constructed examination via the utility search routines has been designed but not implemented.

The TCAP system was originally designed to be operated from an alphanumeric display, hence the mode display, function display terminology, but the present implementation was accomplished using teletypes. Alphanumeric displays have been acquired and many user actions will be changed from typed in responses to menu selections via a cursor. These displays will relieve the user of the major portion of the typing load and make the system a great deal easier to use.

Some observations

The TCAP design goals of flexibility, capability and ease of use produced a conflicting set of software requirements. These requirements combined with the fact that the operating system of the computer forced one to treat all drum files as if they were magnetic tapes resulted in a challenging design problem. The requirement for providing the user with computer based equivalents of present capabilities was solved through the use of cascaded drivers and multiple levels of utility routines. Such a scheme enables the drivers to be concerned with operational logic and the utility routines with performing the functions. The use of multiple levels of utility routines provided functional isolation that simplified the structure of the programs. The final TCAP program was highly modular, hierarchical in structure and quite compact.

The use of relative addressing in conjunction with the character oriented file records and a header scheme proved to be advantageous. The approach makes transferring TCAP to other computers an easy task. Hopefully, the only conversion problem will be adjusting the FORTRAN A formats to the target computer. A significant feature of the approach is that record layouts within files are defined at run time rather than at compile time. The practical effect is that each instructor can tailor the number of sections within a record and their size to suit his own needs. Thus, the item, statistics, and test files can be unique to a given user. TCAP modifies its internal file manipulations to process the record specifications it receives. Such flexibility is important in the university setting where each instructor feels his instructional procedures are unique.

One consequence of the high degree of operational flexibility and the range of capabilities provided is that housekeeping within TCAP is extensive. A good example of this housekeeping occurs when the File Maintenance routine updates the item files from the item analysis results file generated by the FORTAP program. Because not all items in the test will have records in the item file, the File Maintenance routine must keep track of them, create records for them, add them to the item file, and inform the user that the records have been added. There are numerous other situations of comparable complexity throughout the TCAP system. Handling them smoothly and efficiently is a difficult task. Because TCAP was implemented on a large computer, such situations were generally handled by creating supplementary drum files and provided working arrays in core. The use of random access files would have greatly simplified many of the internal housekeeping problems.

On the basis of the author's experience with the design and implementation of the TCAP system one salient conclusion emerges. Such programs must be designed as complete software systems. To attempt to design them in a sequential fashion and implement them piecemeal is folly. The total system needs to be thought through very carefully and the possible interactions explored. If provision is to be made for future, but undefined, extensions, the structure of the program and the files must be kept simple to reduce the interaction effects of such enhancements. It appears to be a characteristic of this area of computer programming that complexity and chaos await your every decision. This caveat is a reflection of the many design iterations that were necessary to achieve the TCAP system. The end product of this process is a system that provides the instructor with an easy to use tool that can be of considerable assistance. Being able to maintain an item bank and assemble tests to meet arbitrary specifications aids one in performing an unavoidable task. To do so quickly and efficiently is worth the investment it takes to convert one's item bank into machine readable form. The TCAP system illustrates again that tasks performed by manual means can often be quite difficult to implement by computer. In the present case a reasonable implementation was achieved by making the system interactive and taking advantage of the capabilities of both man and machine.

Measurement of computer systems—An introduction

by ARNOLD F. GOODMAN

McDonnell Douglas Astronautics Company
Huntington Beach, California

NEED FOR MEASUREMENT

Computer systems have become indispensable to the advancement of management, science and technology. They are widely employed by academic, business and governmental organizations. Their contribution to today's world is significant in terms of both quantity and quality.

This significant growth of computer utilization has been accompanied by a similar growth in computer technology. Faster computers with larger memories and more flexible input and output have been introduced, one after another. Interactive, multiprocessing, multiprogramming, realtime and timesharing have been transformed from catchy slogans into costly reality—or at least, partial reality.

In addition, computer science has come into being, and has made great progress from an art toward a science. Departments of computer science have appeared within many colleges and universities. A new profession has been created and is attempting to mature.

These three areas of phenomenal growth—computer utilization, computer technology and computer science—have produced the requirement for a new field, measurement of computer systems. In an atmosphere of escalating computer cost and increasing budget scrutiny, measurement provides a bridge between design promises and operational performance. This function of measurement is complemented by the traditional need for measurement of any art in search of a science.

ACTIVITY INVOLVING MEASUREMENT

A limited survey was conducted of the 1960-1970 literature on measurement of computer systems. This survey included all *Proceedings of Spring Joint Computer Conferences, Proceedings of Fall Joint Computer Conferences, Journals of the Association for Computing Machinery* and *Communications of the ACM*, as well as selected *Proceedings of ACM National Conferences* and *Proceedings of Conferences on Application of Simulation.* The resulting personal bibliography and the unpublished bibliographies of Bell[1], Miller[2] and Robinson[3]—each with its own bias and deficiency—were utilized to obtain an initial indication of pioneer activity involving measurement.

Measurement of computer systems was presaged by Herbst, Metropolis and Wells[4] in 1945, Shannon[5] in 1948, Hamming[6] in 1950 and Grosch[7] in 1953. Bagley,[8] Black,[9] Codd,[10] Fein,[11] Flores,[12] Maron[13] and Nagler[14] published articles concerning it during 1960. These were followed in 1961 with the related contributions of Barton,[15] Flores,[16] Gordon,[17] Gurk and Minker,[18] Hosier,[19] and Jaffe and Berkowitz.[20] During 1962, there were pertinent papers by Adams,[21] Baldwin, Gibson and Poland,[22] Dopping,[23] Gosden and Sisson,[24] Hibbard,[25] Patrick,[26] Sauder,[27] Simonsen[28] and Smith.[29]

Many of the concepts and techniques which were developed for defense and space systems—whose focal point was hardware rather than software—are also applicable to computer systems. The system design, development and testing sequence was perfected by the late 1950's. Since the early 1960's, system verification, validation, and cost and effectiveness evaluation have been prevalent. The adaptation of these concepts and techniques to measurement of computer systems—especially software—is not as simple as system specialists tend to believe, yet not as difficult as software specialists tend to believe.

In the middle 1960's, such concepts and techniques began to be applied to the selection and evaluation of computer systems, and to software as well as hardware. Ratynski,[30] Searle and Neil,[31] Liebowitz[32] and Piligian and Pokorney[33] describe the Air Force and National Aeronautics and Space Administration (NASA) adaptation of their system acquisition procedures to software acquisition. Attention then shifted to measurement of computer system performance, with a corresponding increase of activity. Sackman[34] discusses computer system development and testing, based upon the Air Force and NASA experience. An important development of the period was the formation of a Hardware Evaluation Committee within SHARE[35] during early 1964, and its evolution into the SHARE Computer Measurement and Evaluation Project[36] during August 1970, which served as a focal point for significant progress.[37]

A preliminary but informative indication of activity involving computer system effectiveness evaluation prior to 1970 appears below. When a comprehensive bibliography on measurement of computer systems is compiled and annotated, the gross characterization of activity given in this paper may be refined and expanded—especially in the area of practical contributions and contributors to measurement. Raw material for that bibliography and characterization may be found in the unpublished bibliographies of Bell,[1] Miller,[2] Robinson[3] and the author mentioned above—as well as a bibliography by Crooke and Minker,[38] one in preparation by Menck,[39] and the selected papers in Hall.[37]

During a keynote address at Computer Science and Statistics: Fourth Annual Symposium on the Interface in September 1970, Hamming coined the name of "compumetrics"—in the spirit of biometrics, econometrics and psychometrics—for measurement of computer systems.[40] It is fitting that the naming of compumetrics occurred at this symposium, since measurement of computer systems is truly a part of the interface—or area of interaction—of computer science and statistics.[41]

Hamming phrased it well when he stated: [40]

> "The director of a computer center is responsible for managing the utilization of large amounts of money, people and resources. Although he has a complex and important statistical problem, his decisions are normally based upon the simplest collection and analysis of data—since he usually knows little statistics beyond such elementary concepts as the mean and variance. His need for statistics involves both the operational performance of his hardware and software, and the environment provided by his organization and users."
>
> "A new discipline that seeks to answer these questions—and that might be called 'compumetrics'—is in the process of evolving. Karl Pearson and R. A. Fisher established themselves by developing novel statistical solutions to significant problems of their time. Compumetrics may well provide contemporary statisticians with many such opportunities."

Workshop sessions on compumetrics followed Hamming's remarks at the Fourth Symposium on the Interface. During these sessions,[40] "there developed a feeling that this symposium marked a beginning which must not be allowed to be an end"—that sessions on compumetrics be scheduled at the Fifth Symposium on the Interface, and that a local steering committee be formed to promote interest in compumetrics.

It is not surprising, therefore, that a Special Interest Committee on Measurement of Computer Systems—SICMETRICS—was initiated within the Los Angeles Chapter of the Association for Computing Machinery during April 1971. SICMETRICS is compiling a bibliography on compumetrics.[39]

There were sessions on computer system models and analysis at the Fifth Annual Princeton Conference on Information Sciences and Systems[42] in March 1971. In April 1971, the ACM Special Interest Group on Operating Systems—SIGOPS—sponsored a Workshop on System Performance Evaluation[43]—with sessions on instrumentation, mathematical models, queuing models, simulation models and performance evaluation. There were sessions on system evaluation and diagnostics at the 1971 Spring Joint Computer Conference[44] during May 1971. This was followed in November 1971 by workshop sessions on compumetrics at the Fifth Symposium on the Interface,[45] by a session on operating system models and measures at the 1971 Fall Joint Computer Conference,[46] and by a Conference on Statistical Methods for Evaluation of Computer Systems Performance[47]—with sessions on general approaches, evaluation of current systems, input analysis, software reliability, system management, design of experiments and regression analysis. During November 1971, the ACM Special Interest Committee on Measurement and Evaluation—SICME—was also formed.

The ACM Special Interest Groups on Programming Languages—SIGPLAN—and on Automata and Computability Theory—SIGACT—sponsored a Conference on Proving Assertions about Programs[48] in January 1972. A Symposium on Effective Versus Efficient Computing[49]—with sessions on responsibility, getting results, implementation, evaluation, education and looking ahead—was held during March 1972, and so was a session on computer system models at the Sixth Annual Princeton Conference on Information Sciences and Systems.[50] In May 1972, there was a session on compumetrics at the 1972 Technical Symposium of the Southern California Region of ACM, and there were sessions on system performance measurement and evaluation at the 1972 Spring Joint Computer Conference.[51] An ACM Special Interest Group on Programming Languages—SIGPLAN—Symposium on Computer Program Test Methods followed during June 1972.[52]

The National Bureau of Standards and ACM are jointly sponsoring a series of workshops and conferences on performance measurement. An informative discussion of many practical aspects of compumetrics is contained in Canning.[53] Finally, the 1972 Fall Joint Computer Conference[54] in December 1972, has coordinated sessions on measurement of computer systems—executive viewpoints, system performance, software validation and reliability, analysis considerations, monitors and their application, and case studies.

Across the Atlantic, a Performance Measurement Specialist Group was organized within the British Computer Society in early 1971. A number of its working groups are functioning on specific projects, and it sponsored a conference in September 1972.

This summary of activity involving measurement of computer systems clearly outlines the growth and increasing importance of compumetrics. Proposal of a structure for compumetrics is, therefore, quite appropriate. The presentation below is general and suggestive, rather than detailed and complete—as is appropriate for an introduction.

STRUCTURE FOR MEASUREMENT

A structure—or framework—is proposed for measurement of computer systems, to serve as a background for both understanding and developing the subject. It provides not only a common set of terms—which may be familiar to some and new to others, but also a guide to the current—as well as potential—extent and content of compumetrics. Such a structure is critical for subjects that have matured and crucial otherwise, whether or not there is universal agreement on detailed portions of it. The conceptual framework for Air Force and NASA acquisition of computer systems[30–34] provides a context in which not only the structure for measurement, but also the structure for effectiveness evaluation, should be considered.

Compumetrics concerns measurement in—internal to—or of—external to—computer systems. As for biometrics, econometrics and psychometrics, this means measurement of a general nature applied to computer systems in a broad sense. A computer system is taken to be a collection of properly related elements, including a computer, which possesses a computing or data handling objective. The structure for compumetrics is described in terms of computer system evolution and computer system operation. Computer system evolution is divided into design, development and testing, and computer system operation is divided into objective, composition and management. A sequence of questions—including the if, why, what, where, when, how much and how of measurement—should be developed and then answered for each element of the structure.

The structure is presented from the viewpoint of a statistician who is knowledgeable about computers, in order to augment Hamming's viewpoint as a computer scientist who is knowledgeable about statistics. In addition, this structure is considerably more comprehensive and definitive than that which is implied by Hamming's original discussion.[40] An outline version of it appeared in Locks.[45]

At present, measurement of computer systems might be characterized as a growing collection of measurements on their way toward a science, and in need of planning and analysis to help them get there. Bell, Boehm and Watson[55] provide an adaptation of the scientific method to performance measurement and improvement of a computer system: from understanding the system and analyzing its operation, through formulating performance improvement hypotheses and analyzing the probable cost-effectiveness of the corresponding modifications, to testing specific hypotheses and implementing the appropriate combinations of modifications—as well as testing the cost-effectiveness of these combinations. As a complement to this approach, the author[56] presents a user's guide to data modeling and analysis—including a perspective for viewing and utilizing such a framework for the collection and analysis of measurements. That paper[56] discusses the sequence of steps which leads from a problem through a solution to its assessment, some aspects of solving problems which should be considered, and an approach to the design and analysis of a complex system through utilization of both experimental and computer simulation data.

Measurement and system evolution

Within this and the following sections, appropriate terms appear in capital letters for emphasis. Such a procedure produces not only clarity of exposition, but also a lack of smoothness, in the resulting text. The advantage of the former is sought, even at the disadvantage of the latter. In addition, words are employed in their usual nontechnical sense.

Computer systems evolve from DESIGN through

DEVELOPMENT to TESTING. For illustrative purposes, we present one partition—from among the many which are possible—of this evolution into more basic components. It is meaningful from both a manager's and a user's point of view. For a given computer system, the accomplishment of more than one component may be occurring simultaneously, and the accomplishment of all components may not be feasible.

The DESIGN of a computer system involves the system what and how. A REQUIREMENTS ANALYSIS ascertains user needs and generates system objectives, and a FUNCTIONAL ANALYSIS translates system objectives into a desired system framework. Then SPECIFICATION SYNTHESIS transforms the objectives and desired framework into desired performance and its description. Finally, STRUCTURE develops system framework from the desired framework, and SIZING infers system size from its framework.

System DEVELOPMENT is concerned with implementing the system what and how. It proceeds from HARDWARE AND SOFTWARE SELECTION—which includes the decision to make or buy, through HARDWARE AND SOFTWARE ACQUISITION—which involves either making or buying—and HARDWARE AND SOFTWARE COMBINATION—which implements the framework in terms of acquired hardware and software, to SOFTWARE PROGRAMMING—which includes the programming of additional software. How well the framework was implemented is then determined by HARDWARE AND SOFTWARE VERIFICATION. Development is completed by SYSTEM DOCUMENTATION to describe the system what and how, and by PROCEDURE DOCUMENTATION to describe the how of system operation and use.

TESTING of a computer system has the objective of assessing how well the system performs. First, system INTEGRATION—which could have been included under development—assembles the hardware, software and other elements into a system. This is followed by system VALIDATION, for ascertaining how well the specifications were implemented and for contributing to quality assurance. COST EVALUATION determines how much the system costs in terms of evolution and operation, and EFFECTIVENESS EVALUATION determines how well the system performs in terms of operational time, quality and impact upon the user. The final step in testing is, of course, OPERATION—performance for the user.

McLean[57] proposes a characterization for the "all-too-true life cycle of a typical EDP system: unwarranted enthusiasm, uncritical acceptance, growing concern, unmitigated disaster, search for the guilty, punishment of the innocent, and promotion of the uninvolved." An excellent discussion of computer system development and testing—whose application should alter this cycle—is provided by Sackman.[34] In addition, measurement was apparently employed in many places within the design, development and testing sequence for the information system of Winbrow.[58]

Where is measurement currently utilized in the system evolution sequence? Measurement is inherently involved in hardware specification synthesis, sizing and cost evaluation. It is employed to a limited extent during hardware requirements analysis and selection, and it emerged in importance as a significant contributor to hardware validation and performance monitoring—which is a portion of effectiveness evaluation. We are only beginning to consider serious and systematic measurement as it concerns software verification, validation, and cost and effectiveness evaluation. In fact, we are beginning to use the same terminology for hardware and software that was used in the early 1960's for defense and space systems—which were predominately noncomputer hardware. "Requirements for AVAILABILITY of Computing System Facilities"[59] provides an excellent example, with its use of reliability, maintainability, repairability and recoverability.

Where should measurement be utilized in the evolution sequence? It probably has an appropriate use in most, if not almost all, components of the sequence. In particular, system verification, validation, and cost and effectiveness evaluation—as well as reliability and its fellow *ilities*[59]—have no real meaning without measurement.

Measurement and system operation

A computer system operation has COMPOSITION and an OBJECTIVE, as well as being subject to MANAGEMENT. As a guide to discussion and thought, a useful—but not unique—division of system operation into more basic elements is now described. A given computer system, however, may not involve all of these elements.

COMPOSITION of a computer system concerns what constitutes the system. The main component, by tradition, has been computer HARDWARE—which may involve input, memory, processing, output, communication or special purpose equipment. Since the means for communicating with that equipment currently costs from one to ten times as much as the hardware, the main component really is SOFTWARE—which may involve input, storage and retrieval, operating, application, simulation, output or communication program packages. The system may also contain

FIRMWARE, which is either soft hardware or hard software—such as a microprogram, and PERSONNEL. How to operate and use the system is covered by the operating PROCEDURE. The system aspects include all two way INTERFACES such as hardware-software, all three way INTERFACES such as firmware-personnel-procedure, all four way INTERFACES such as hardware-software-personnel-procedure, and the five way INTERFACE of hardware-software-firmware-personnel-procedure.

What the computer system does primarily—although it may do many things concurrently or sequentially—is the system OBJECTIVE. DATA MANAGEMENT emphasizes storage and retrieval of data by the system. Operating upon data by the system is the focus of DATA PROCESSING. COMMAND AND CONTROL stresses input and output of data by the system, and decisions aided by the system.

As observed by Boehm, an alternative view is that all three types of systems aid the making of decisions: data management systems provide the least aid, data processing systems provide more aid, and command and control systems provide the most aid. The distinction among these also depends upon what the environment is and who the user is—data management or command and control systems are frequently called information systems. In addition, the same system— or a portion of it—might frequently be utilized for more than one objective.

Computer system MANAGEMENT involves system administration and supervision. PLANNING is projecting the system's future. Getting operations together and focused constitutes COORDINATION, and keeping operations together and directed constitutes CONTROL. REVIEW provides an assessment of the past and present, while TRAINING provides system operators. Finally, USER INTERACTION concerns system calibration and acceptance by the user.

Measurement has traditionally been employed on computer hardware and personnel, has begun to be employed on software and firmware, and may someday be employed on procedure and interfaces. It has been applied in data management and data processing, but should also be applied in command and control. As for management in general, measurement is only beginning to be utilized in computer system planning, coordination, control, review, training and user interaction.

STRUCTURE FOR EFFECTIVENESS EVALUATION

Consideration of the need for, activity involving, and structure for measurement implies that an important unsolved problem for the 1970's is the evaluation of computer system effectiveness. That this is true for library information systems is explicitly stated in a recent report by the National Academy of Sciences Computer Science and Engineering Board,[60] and that it is true for computer systems in general is implicitly stated in a recent report by GUIDE International.[59] As Maclean observed,[57] we are like Oscar Wilde's cynic: "A man who knows the price of everything, and the value of nothing."

Effectiveness evaluation determines how well the system performs in terms of operational time, quality and impact upon the user. It has both an internal or inwardly oriented aspect—which determines how well the system responds to any need, and is more efficiency than effectiveness—and an external or outwardly oriented aspect—which determines how well the system responds to the actual need, and is truly effectiveness. The point of view that is taken as to what effectiveness is and how it should be evaluated is also extremely important. Viewpoints of the user and his management should be considered, as well as viewpoints of the system and its management. In terms of both aspects and viewpoints, effectiveness evaluation is much broader than mere performance measurement.

Evaluating the impact of the system upon a user is essentially the reverse of system design or selection, which evaluates the impact of the user upon a potential or real system. In order to accomplish this, it is necessary to evaluate how well the promises of system design or selection are fulfilled by system operation. An informative, as well as interesting, exercise would be the real impact evaluation of applications such as those surveyed in 1965 by Rhodes,[61] Ramo,[62], Gerard,[63] Maloney,[64] McBrier,[65] Merkin and Long,[66] Gates and Pickering,[67] Ward,[68] Baran[69] and Schlager.[70]

Based upon Air Force and NASA experience, Sackman[34] provides a thorough treatment of computer system development and testing. This treatment includes:

- A survey of system engineering, human factors, software and operations research points of view on testing and evaluation—all of which are implicitly oriented inwardly toward the system, rather than outwardly toward the user.
- A description of test levels, objectives, phasing within development and operation, approach and chronology.
- A discussion of the analogy between scientific method and system development—during which, a sequence of increasingly specific hypotheses is posed and tested, as the implicit promises of

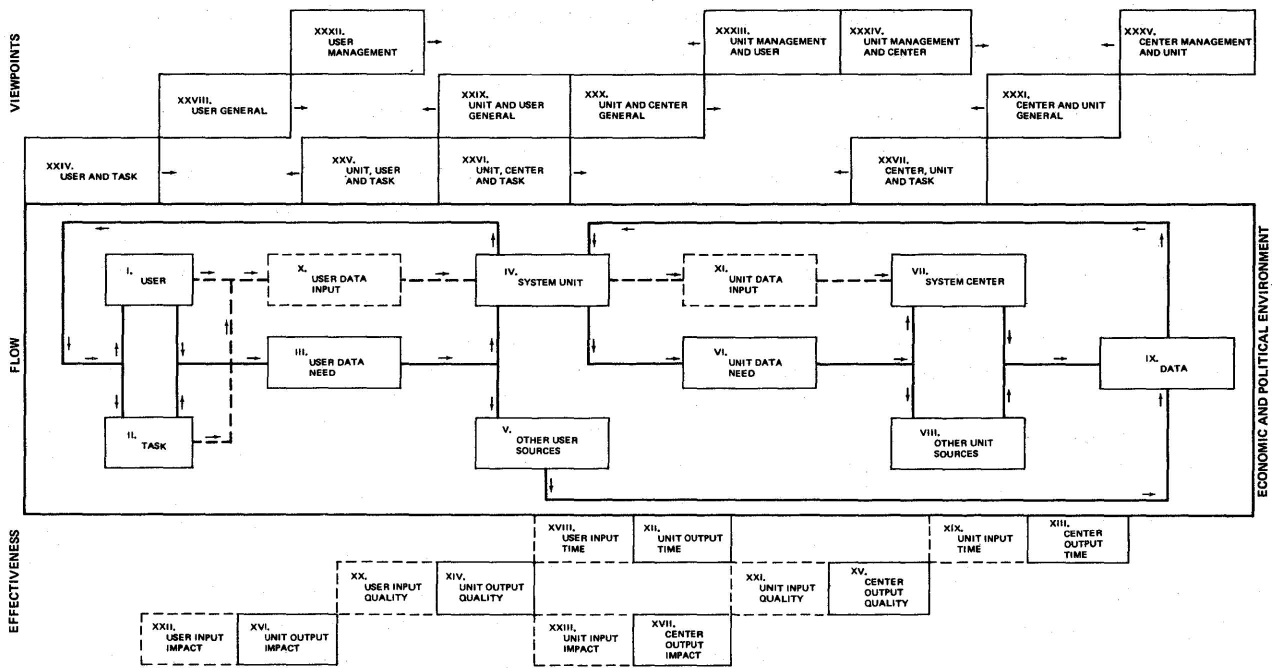


Figure 1—Structure for evaluation of data management or command and control system effectiveness

design become explicit promises during development and explicit performance during operation.

- A summary of the philosophical roots of this analogy and approach.
- A short bibliography.

It constitutes an excellent contribution to effectiveness evaluation, as well as a firm foundation for the framework of Bell, Boehm and Watson,[55] but more is needed. In addition, almost all library system effectiveness evaluation has been centered around—if not actually restricted to—variations of two simple ratios, called relevance and recall. And Fingings 1 and 2 in the National Academy of Sciences report[60] state that much more is needed.

The complexity and importance of effectiveness evaluation combine to require a significantly broader and deeper, as well as more meaningful, structure. Most of the significance and ultimate payoff associated with computer systems involve the external environment and aspects of the system, from various points of view. Despite that fact, the preponderance of effectiveness evaluation has not focused upon such aspects from the appropriate points of view.[34,71–79]

A structure for computer system effectiveness evaluation is proposed, as both a step toward fulfilling that need and an elaboration of the structure for compumetrics. Figure 1 contains a general version of the structure for data management or command and control systems, and Figure 2 contains a general version of the structure for data processing systems. The graphic presentations of the figures are complemented by the corresponding verbal descriptions—which employ words in their usual nontechnical sense. Effectiveness evaluation of a computer system might require a combination of the structures in Figures 1 and 2, since the system might frequently be utilized for more than one objective. In addition, the entire structure might not be of interest for a given system.

An initial indication of activity involving computer system effectiveness evaluation is then summarized. Finally, selected papers that illustrate such activity are briefly discussed. This summary and discussion serve as a background against which to view the proposed structures.

Evaluation of data management or command and control systems

In Figure 1, there are three main categories of characteristics—FLOW, EFFECTIVENESS and VIEWPOINTS—all of which reside within an ECONOMIC AND POLITICAL ENVIRONMENT. FLOW characteristics (I-XI) involve the flow of data and need for data, from a user and his task through the system unit and center back to the user and his task. Those characteristics (XII-XXIII) which describe how well the flow of data satisfies the need for data—

both internal and external to the system—comprise EFFECTIVENESS. VIEWPOINTS contain the various points of view (XXIV-XXXV) regarding the flow and its effectiveness. All of these characteristics are embedded within an ECONOMIC AND POLITICAL ENVIRONMENT, whose influence is sometimes explicit and sometimes implicit yet always present.

A USER (I) of the system and a TASK (II) which he is performing jointly generate a need for data, called USER DATA NEED (III). To satisfy this need, the user contacts either the appropriate outlet of the system—SYSTEM UNIT (IV)—or other sources for data—OTHER USER SOURCES (V). The unit essentially becomes a user now and contacts either the SYSTEM CENTER (VII) or OTHER UNIT SOURCES (VIII), in order to satisfy its UNIT DATA NEED (VI). DATA (IX) is then output by the system or other sources to the user for performance of his task. Finally, there may also be USER DATA INPUT (X)—such as data generated by the user in his task or by user management regarding an impending change in its basic need—by the user to the unit, and UNIT DATA INPUT (XI)—such as data generated by the unit or by unit management regarding an impending change in its basic need—by the unit to the system.

Operational characteristics of the unit and center in terms of time—how quickly or how often—are grouped under UNIT OUTPUT TIME (XII) and CENTER OUTPUT TIME (XIII), those in terms of quality—how well or how completely—are grouped under UNIT OUTPUT QUALITY (XIV) and CENTER OUTPUT QUALITY (XV), and those in terms of impact—how responsively or how significantly—are grouped under UNIT OUTPUT IMPACT (XVI) and CENTER OUTPUT IMPACT (XVII). Time characteristics emphasize the internal aspects of the system and impact characteristics emphasize the external aspects of the system, while quality characteristics emphasize both the internal and external aspects of the system. In addition, time is the easiest to measure objectively as well as the least meaningful quality is more difficult to measure objectively than time and less difficult to measure objectively than impact, as well as more meaningful than time and less meaningful than impact . . . impact is the most difficult to measure objectively as well as the most meaningful. Effectiveness may be viewed as the average, over all users and tasks, of the effectiveness for specific user and task combinations.

There may also be USER INPUT TIME (XVIII) and UNIT INPUT TIME (XIX)—to indicate how quickly or how often the user inputs data to the unit and the unit inputs data to the center, USER INPUT QUALITY (XX) and UNIT INPUT QUALITY (XXI)—to indicate how well or how completely these

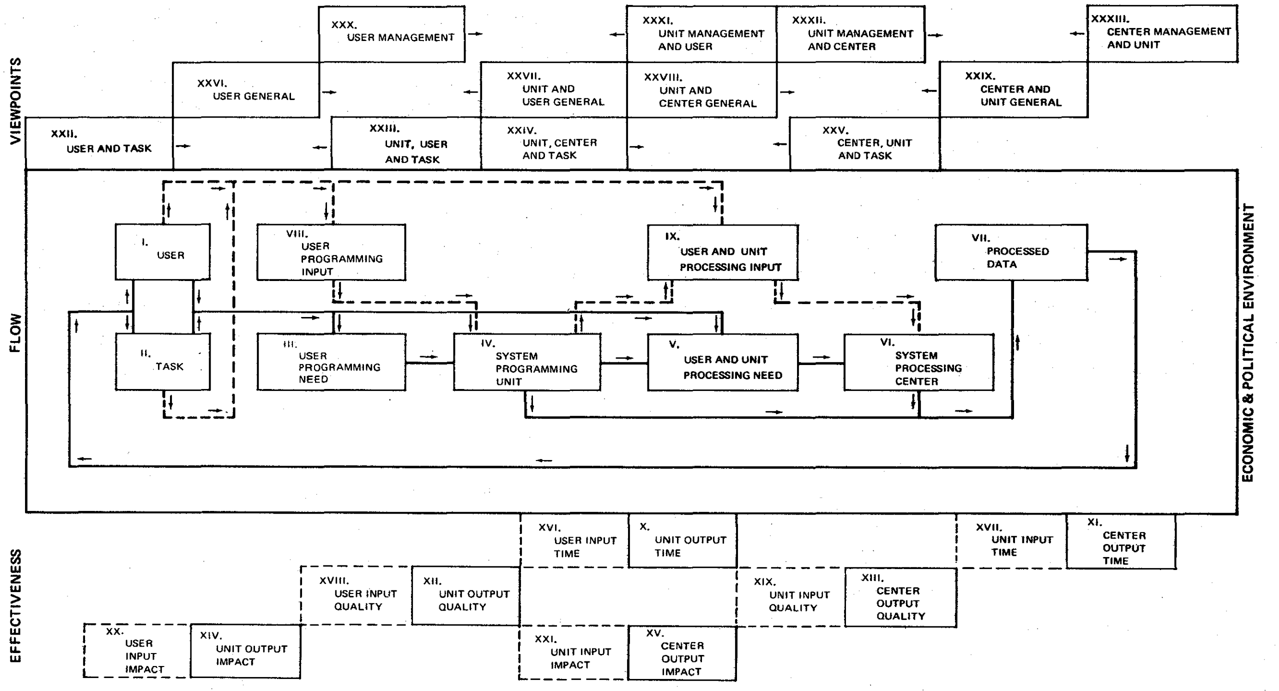

VIEWPOINTS
XXX. USER MANAGEMENT
XXXI. UNIT MANAGEMENT AND USER
XXXII. UNIT MANAGEMENT AND CENTER
XXXIII. CENTER MANAGEMENT AND UNIT
XXVI. USER GENERAL
XXVII. UNIT AND USER GENERAL
XXVIII. UNIT AND CENTER GENERAL
XXIX. CENTER AND UNIT GENERAL
XXII. USER AND TASK
XXIII. UNIT, USER AND TASK
XXIV. UNIT, CENTER AND TASK
XXV. CENTER, UNIT AND TASK
FLOW
I. USER
VIII. USER PROGRAMMING INPUT
IX. USER AND UNIT PROCESSING INPUT
VII. PROCESSED DATA
II. TASK
III. USER PROGRAMMING NEED
IV. SYSTEM PROGRAMMING UNIT
V. USER AND UNIT PROCESSING NEED
VI. SYSTEM PROCESSING CENTER
ECONOMIC & POLITICAL ENVIRONMENT
EFFECTIVENESS
XVI. USER INPUT TIME
X. UNIT OUTPUT TIME
XVII. UNIT INPUT TIME
XI. CENTER OUTPUT TIME
XVIII. USER INPUT QUALITY
XII. UNIT OUTPUT QUALITY
XIX. UNIT INPUT QUALITY
XIII. CENTER OUTPUT QUALITY
XX. USER INPUT IMPACT
XIV. UNIT OUTPUT IMPACT
XXI. UNIT INPUT IMPACT
XV. CENTER OUTPUT IMPACT

Figure 2—Structure for evaluation of data processing system effectiveness

were accomplished, and USER INPUT IMPACT (XXII) and UNIT INPUT IMPACT (XXIII)—to indicate how responsively or how significantly these were accomplished. In this case, the user is serving the system and the above roles are reversed. Internal aspects of the user are focused upon by time and external aspects of the user are focused upon by impact, while both internal and external aspects of the user are focused upon by quality.

What we mean by effectiveness, as well as how we evaluate it, will vary according to our point of view. The task specific viewpoint of the user toward the unit is USER AND TASK (XXIV), that of the unit toward the user is UNIT, USER AND TASK (XXV), that of the unit toward the center is UNIT, CENTER AND TASK (XXVI), and that of the center toward the unit is CENTER, UNIT AND TASK (XXVII). USER GENERAL (XXVIII), UNIT AND USER GENERAL (XXIX), UNIT AND CENTER GENERAL (XXX), and CENTER AND UNIT GENERAL (XXXI) represent general viewpoints of the user for the unit, the unit for the user, the unit for the center, and the center for the unit. Finally, the viewpoint of user management toward the unit constitutes USER MANAGEMENT (XXXII), that of unit management toward the user constitutes UNIT MANAGEMENT AND USER (XXXIII), that of unit management toward the center constitutes UNIT MANAGEMENT AND CENTER (XXXIV), and that of center management toward the unit constitutes CENTER MANAGEMENT AND UNIT (XXXV). Internal aspects of the system are stressed in center viewpoints and external aspects of the system are stressed in user viewpoints, while both internal and external aspects of the system are stressed in unit viewpoints. Task specific viewpoints are the easiest to measure objectively, general viewpoints are more difficult to measure objectively than task specific viewpoints and less difficult to measure objectively than management viewpoints, and management viewpoints are the most difficult to measure objectively—the meaningfulness of these depends, of course, upon point of view.

Evaluation of data processing systems

Figure 2 contains the characteristics of FLOW (I-IV), EFFECTIVENESS (X-XXI) and VIEWPOINTS (XXII-XXXIII)—all being surrounded by an ECONOMIC AND POLITICAL ENVIRONMENT. Since it differs from Figure 1 only in terms of the basic flow for data and need, a brief description is now presented.

A USER (I) and his TASK (II) jointly generate USER PROGRAMMING NEED (III) or USER PROCESSING NEED (V). To satisfy this need, the user contacts the SYSTEM PROGRAMMING UNIT (IV) or SYSTEM PROCESSING CENTER (VI)—which is also contacted to satisfy USER AND UNIT PROCESSING NEED (V). PROCESSED DATA (VII) is then output to the user for performance of his task. There may also be USER PROGRAMMING INPUT (VIII) by the user to the unit, or USER AND UNIT PROCESSING INPUT (IX) by the user and unit to the center.

Operational characteristics of the unit and center are grouped under UNIT OUTPUT TIME (X) and CENTER OUTPUT TIME (XI), UNIT OUTPUT QUALITY (XII) and CENTER OUTPUT QUALITY (XIII), and UNIT OUTPUT IMPACT (XIV) and CENTER OUTPUT IMPACT (XV). There may also be USER INPUT TIME (XVI) and UNIT INPUT TIME (XVII), USER INPUT QUALITY (XVIII) and UNIT INPUT QUALITY (XIX), and USER INPUT IMPACT (XX) and UNIT INPUT IMPACT (XXI).

Task specific viewpoints are those of USER AND TASK (XXII), UNIT, USER AND TASK (XXIII), UNIT, CENTER AND TASK (XXIV), and CENTER, UNIT AND TASK (XXV). USER GENERAL (XXVI), UNIT AND USER GENERAL (XXVII), UNIT AND CENTER GENERAL (XXVIII), and CENTER AND UNIT GENERAL (XXIX) represent general viewpoints. Finally, management viewpoints are given by USER MANAGEMENT (XXX), UNIT MANAGEMENT AND USER (XXXI), UNIT MANAGEMENT AND CENTER (XXXII), and CENTER MANAGEMENT AND UNIT (XXXIII).

Some modification and considerable refinement may be required to employ one of these structures on an actual computer system. The structures do, however, indicate important considerations for evaluating the effectiveness of a computer system. In addition, they are considerably more comprehensive than current structures, and provide a guide toward their own modification and refinement.

Activity involving evaluation

This introduction to compumetrics concludes with an initial indication of activity involving computer system effectiveness evaluation prior to 1970, and a brief description of selected papers which illustrate the activity. That indication and description provide a context in which to consider the structures given above.

Utilizing the unpublished bibliographies of Bell,[1] Miller,[2] and Robinson[3] and the author, each processing its own bias and deficiency, a preliminary characterization of effectiveness evaluation activity before 1970 was obtained. Those pioneering papers that appeared prior to 1963 and treated the general topic were included, but those papers that emphasized mathematical modeling or computer simulation—the majority of which were more concerned with mathematics than with measurement—were not included.

There were 234 separate references remaining after duplicate listings within these bibliographies were eliminated. The number (and approximate percentage) of documents by year were:

- 1945—1 (0%)
- 1948—1 (0%)
- 1950—1 (0%)
- 1953—1 (0%)
- 1960—7 (3%)
- 1961—6 (2%)
- 1962—9 (4%)
- 1963—7 (3%)
- 1964—14 (6%)
- 1965—8 (3%)
- 1966—13 (6%)
- 1967—23 (10%)
- 1968—31 (14%)
- 1969—62 (27%)
- 1970—50 (22%)

These numbers and percentages are, of course, affected by all pioneering papers having been counted at the lower end and by some recent papers having possibly been missed at the upper end. Nevertheless, they do exhibit a general trend in the variation of activity over the period. A serious characterization of such activity awaits the compilation and annotation of a comprehensive bibliography on measurement of computer systems—by categories in the structures for measurement and effectiveness evaluation, as well as by year.

An elementary structure for evaluation of command and control system effectiveness—in its external form as well as its internal form—is provided by Edwards.[71] Both Rosin[72] and Bryan[73] consider time and quality characteristics of data processing system performance for a large variety of users, the former on a batchprocessing system and the latter on a timesharing system. Five experiments for comparing the performance of a timesharing system with that of a batchprocessing system—Gold,[74] Sackman, Erikson and Grant,[75] Schatzoff, Tsao and Wiig,[76] and Smith[77]—are summarized by Sackman:[78]

- All five employ computer time and some measure of man time.
- All five employ some measure of program quality.
- Gold employs three additional measures of quality, and Smith employs one additional measure of quality.
- Gold and Schatzoff, Tsao and Wiig employ a measure of cost.
- All five employ—in an implicit, rather than explicit, manner—both system and user viewpoints.

Finally, Shemer and Heying[79] include both internal and external aspects of effectiveness in the design model for a system, which is to perform timesharing as well as batchprocessing—and then compare operational system data with the design model.

ACKNOWLEDGMENTS

The critical review of this paper and constructive suggestions for its improvement by Thomas Bell, Barry Boehm, Richard Hamming, Robert Patrick, Harold Petersen and Louis Robinson are gratefullv acknowledged.

REFERENCES

1 T E BELL
Computer system performance bibliography
Unpublished

2 E F MILLER JR
Bibliography on techniques of computer performance analysis
Unpublished

3 L ROBINSON
Bibliography on data processing performance evaluation
Unpublished

4 E H HERBST N METROPOLIS N B WELLS
Analysis of problem codes on the MANIAC
Mathematical Tables and Other Aids to Computation
Vol 9 No 49 1945 pp 14-20

5 C E SHANNON
A mathematical theory of communication
Bell System Technical Journal Vol 27 1948 p 379

6 R W HAMMING
Error detecting and error correcting codes
Bell System Technical Journal Vol 29 1950 p 147

7 H R J GROSCH
High speed arithmetic: The digital computer as a research tool
Journal of the Optical Society of America Vol 43 No 4
1953 pp 306-310

8 P R BAGLEY
Item 2 of two think pieces: Establishing a measure of

capability of a data processing system
Communications of the ACM Vol 3 No 1 1960 p 1

9 A J BLACK
SAVDAT: A routine to save input data in simulator tape format
Report FN-GS-151 System Development Corporation 1960

10 E F CODD
Multiprogram scheduling: Parts I-IV
Communications of the ACM Vol 3 Nos 6 and 7 1960 pp 347-350 and 413-418

11 L FEIN
A figure of merit for evaluating a control computer system
Automatic Control 1960

12 I FLORES
Computer time for address calculation sorting
Journal of the Association for Computing Machinery Vol 7 No 4 1960 pp 389-409

13 M E MARON J L KUHNS
On relevance probabilistic indexing and information retrieval
Journal of the Association for Computing Machinery Vol 7 No 3 1960 pp 389–409

14 H NAGLER
An estimation of the relative efficiency of two internal sorting methods
Communications of the ACM Vol 3 No 11 1960 pp 618-620

15 R S BARTON
A new approach to the functional design of a digital computer
Proceedings of 1961 Fall Joint Computer Conference AFIPS Press 1961 pp 393-396

16 I FLORES
Analysis of internal computer sorting
Journal of the Association for Computing Machinery Vol 8 No 1 1961 pp 41-80

17 G GORDON
A general purpose systems simulation program
Proceedings of 1961 Spring Joint Computer Conference AFIPS Press 1961 pp 87-98

18 H M GURK J MINKER
The design and simulation of an information processing system
Journal of the Association for Computing Machinery Vol 8 No 2 1961 pp 260-271

19 W A HOSIER
Pitfalls and safeguards in real-time digital systems with emphasis on programming
IRE Transactions on Engineering Management 1961

20 J JAFFE M I BERKOWITZ
The development and uses of a functional model in the simulation of an information-processing system
Report SP-584 System Development Corporation 1961

21 C W ADAMS
Grosch's law repealed
Datamation Vol 8 No 7 1962 pp 38-39

22 F R BALDWIN W B GIBSON C B POLAND
A multiprocessing approach to a large computer system
IBM Systems Journal Vol 1 No 1 1962 pp 64-76

23 O DOPPING
Test problems used for evaluation of computers
BIT Vol 2 No 4 1962 pp 197-202

24 J A GOSDEN R C SISSON
Standardized comparisons of computer performance
Proceedings of 1962 IFIPS Congress 1962 pp 57-61

25 T N HIBBARD
Some combinatorial properties of certain trees with applications to searching and sorting
Journal of the Association for Computing Machinery Vol 9 No 1 1962 pp 13-28

26 R L PATRICK
Let's measure our own performance
Datamation Vol 8 No 6 1962

27 R L SAUDER
A general test data generator for COBOL
Proceedings of 1962 Spring Joint Computer Conference AFIPS Press 1962 pp 371-324

28 R H SIMONSEN
Simulation of a computer timing device
Communications of the ACM Vol 5 No 7 1962 p 383

29 E C SMITH
A directly coupled multiprocessing system
IBM Systems Journal Vol 2 No 3 1962 pp 218-229

30 M V RATYNSKI
The Air Force computer program acquisition concept
Proceedings of 1967 Spring Joint Computer Conference AFIPS Press 1967 pp 33-44

31 L V SEARLE G NEIL
Configuration management of computer programs by the Air Force: Principles and documentation
Proceedings of 1967 Spring Joint Computer Conference AFIPS Press 1967 pp 45-49

32 B H LIEBOWITZ
The technical specification—Key to management control of computer programming
Proceedings of 1967 Spring Joint Computer Conference AFIPS Press 1967 pp 51-59

33 M S PILIGIAN J C POKORNEY
Air Force concepts for the technical control and design verification of computer programs
Proceedings of 1967 Spring Joint Computer Conference AFIPS Press 1967 pp 61-66

34 H SACKMAN
Computers system science and evolving society
John Wiley & Sons Inc 1967

35 *Proceedings of SHARE XX*III
Share Inc 1969

36 *Proceedings of SHARE XXXV*
Share Inc 1970

37 G HALL Editor
Computer measurement and evaluation: Selected papers from the SHARE project
SHARE Inc 1972

38 S CROOKE J MINKER
KWIC index and bibliography on computer systems simulation and evaluation
Computer Science Center University of Maryland 1969

39 H R MENCK Editor
Bibliography on measurement of computer systems
ACM Los Angeles Chapter Special Interest Committee on Measurement of Computer Systems Unpublished

40 A F GOODMAN Editor
Computer science and statistics: Fourth annual symposium on the interface—An interpretative summary
Western Periodicals Company 1971

41 A F GOODMAN
The interface of computer science and statistics
Naval Research Logistics Quarterly Vol 18 No 2 1971 pp 215-229

42 M E VAN VALKENBURG et al Editors
Proceedings of fifth annual Princeton conference on information sciences and systems
Princeton University 1971

43 U O GAGLIARDI Editor
Workshop on system performance evaluation
ACM Special Interest Group on Operating Systems 1971

44 *Proceedings of 1971 Spring Joint Computer Conference*
AFIPS Press 1971

45 M O LOCKS Editor
Proceedings of computer science and statistics: Fifth annual symposium on the interface
Western Periodicals Company 1972

46 *Proceedings of 1971 Fall Joint Computer Conference*
AFIPS Press 1971

47 W F FREIBERGER Editor
Statistical computer performance evaluation
Academic Press 1972

48 J M ADAMS J B JOHNSON R H STARKS Editors
Proceedings of an ACM conference on proving assertions about programs
ACM Special Interest Groups on Programming Languages and on Automata and Computability Theory 1972

49 F GRUENBERGER Editor
Effective versus efficient computing
Publisher to be selected

50 M E VAN VALKENBURG et al Editors
Proceedings of sixth annual Princeton conference on information sciences and systems
Princeton University 1972

51 *Proceedings of 1972 Spring Joint Computer Conference*
AFIPS Press 1972

52 W C HETZEL Editor
Program testing methods
Prentice-Hall Inc 1972

53 R G CANNING Editor
Savings from performance monitoring
EDP Analyzer Vol 10 No 9 1972

54 *Proceedings of 1972 Fall Joint Computer Conference*
AFIPS Press 1972

55 T E BELL B W BOEHM R A WATSON
Framework and initial phases for computer performance improvement
Proceedings of 1972 Fall Joint Computer Conference AFIPS Press 1972

56 A F GOODMAN
Data modeling and analysis for users—A guide to the perplexed
Proceedings of 1972 Fall Joint Computer Conference AFIPS Press 1972

57 E R MacLEAN
Assessing returns from the data processing investment
Effective versus Efficient Computing Publisher to be selected (see 49)

58 J H WINBROW
A large-scale interactive administrative system
IBM Systems Journal Vol 10 No 4 1971 pp 260-282

59 *Requirements for AVAILABILITY of computing facilities*
User Strategy Evaluation Committee GUIDE International Corporation 1970

60 *Libraries and information technology*
Information Systems Panel Computer Science and Engineering BoardNational Academy of Sciences 1972

61 I RHODES
The mighty man-computer team
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 1-4

62 S RAMO
The computer and our changing society
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 5-10

63 R W GERARD
Computers and education
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 11-16

64 J V MALONEY JR
Computers: The physical sciences and medicine
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 17-19

65 C R McBRIER
Impact of computers on retailing
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 21-25

66 W I MERKIN R J LONG
The application of computers to domestic and international trade
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 27-31

67 C R GATES W H PICKERING
The role of computers in space exploration
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 33-35

68 J A WARD
The impact of computers on the government
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 37-44

69 P BARAN
Communication computers and people
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 45-50

70 K J SCHLAGER
The impact of computers on urban transportation
Proceedings of 1965 Fall Joint Computer Conference Part 2 AFIPS Press 1965 pp 51-55

71 N P EDWARDS
On the evaluation of the cost-effectiveness of command and control systems
Proceedings of 1964 Spring Joint Computer Conference AFIPS Press 1964 pp 211-218

72 R F ROSIN
Determining a computing center environment
Communications of the ACM Vol 8 No 7 1965 pp 463-468

73 G E BRYAN
JOSS: 20,000 hours at a console—A statistical evaluation
Proceedings of 1967 Fall Joint Computer Conference AFIPS Press 1967 pp 769-777

74 M GOLD
Time-sharing and batch-processing: An experimental comparison of their values in a problem-solving situation
Communications of the ACM Vol 12 No 5 1969 pp 249-259

75 H SACKMAN W J ERIKSON E E GRANT
Exploratory experimental studies comparing online and offline programming performance
Communications of the ACM Vol 11 No 1 1968 pp 3-11

76 M SCHATZOFF R TSAO R WIIG
An experimental comparison of time sharing and batch processing
Communications of the ACM Vol 10 No 5 1967 pp 261-265

77 L B SMITH
A comparison of batch processing and instant turnaround
Communications of the ACM Vol 10 No 8 1967 pp 495-500

78 H SACKMAN
Time-sharing versus batch-processing: The experimental evidence
Proceedings of 1968 Spring Joint Computer Conference AFIPS Press 1968 pp 1-10

79 J E SHEMER D W HEYING
Performance modeling and empirical measurements in a system designed for batch and time-sharing users
Proceedings of 1969 Fall Joint Computer Conference AFIPS Press 1969 pp 17-26

A highly parallel computing system for information retrieval*

by BEHROOZ PARHAMI

University of California
Los Angeles, California

INTRODUCTION

The tremendous expansion in the volume of recorded knowledge and the desirability of more sophisticated retrieval techniques have resulted in a need for automated information retrieval systems. However, the high cost, in programming and running time, implied by such systems has prevented their widespread use. This high cost stems from a mismatch between the problem to be solved and the conventional architecture of digital computers, optimized for performing serial operations on fixed-size arrays of data.

It is evident that programming and processing costs can be reduced substantially through the use of special-purpose computers, with parallel-processing capabilities, optimized for non-arithmetic computations. This is true because the most common and time-consuming operations encountered in information retrieval applications (e.g., searching and sorting) can make efficient use of parallelism.

In this paper, a special-purpose highly parallel system is proposed for information retrieval applications. The proposed system is called RAPID, Rotating Associative Processor for Information Dissemination, since it is similar in function to a conventional byte-serial associative processor and uses a rotating memory device. RAPID consists of an array processor used in conjunction with a head-per-track disk or drum memory (or any other circulating memory). The array processor consists of a large number of identical cells controlled by a central unit and essentially acts as a filter between the large circulating memory and a central computer. In other words, the capabilities of the array processor are used to search and mark the file. The relevant parts of the file are then selectively processed by the central computer.

PARALLELISM AND INFORMATION RETRIEVAL

Information retrieval may be defined as selective recall of stored knowledge. Here, we do not consider information retrieval systems in their full generality but restrict ourselves to reference and document retrieval systems. Reference (document) retrieval is defined as the selection of a set of references (documents) from a larger collection according to known criteria.

The processing functions required for information retrieval are performed in three phases:

1. Translating the user query into a set of search specifications described in machine language.
2. Searching a large data base and selecting records that satisfy the search criteria.
3. Preparing the output; e.g., formatting the records, extracting the required information, and so on.

Of these three phases, the second one is by far the most difficult and time-consuming; the first one is straightforward and the third one is done only for a small set of records.

The search phase is time-consuming mainly because of the large volumes of information involved since the processing functions performed are very simple. This suggests that the search time may be reduced by using array processors. Array processing is particularly attractive since the search operations can be performed as sequences of very simple primitive operations. Hence, the structure of each processing cell can be made very simple which in turn makes large arrays of cells economically feasible.

Associative memories and processors constitute a special class of array processors, with a large number of small processing elements, which can perform simple pattern matching operations. Because of these desirable characteristics, several proposals have been made for

* This research was supported by the U.S. Office of Naval Research, Mathematical and Information Sciences Division, Contract No. N00014-69-A-0200-4027, NR 048-129.

using associative devices in information retrieval applications.

Before proceeding to review several attempts in this direction, it is appropriate to summarize some properties of an ideal information retrieval system to provide a basis for evaluating different proposals.

P1. *Storage medium:* Large-capacity storage is used which has modular growth and low cost per bit.
P2. *Record format:* Variable-length records are allowed for flexibility and storage efficiency.
P3. *Search speed:* Fast access to a record is possible. The whole data base can be searched in a short time.
P4. *Search types:* Equal-to, greater-than, less-than, and other common search modes are permitted.
P5. *Logical search:* Combination of search results is possible; e.g., Boolean and threshold functions of simple search results.

Some proposals[1-3] consider using conventional associative memories with fixed word-lengths and, hence, do not satisfy P2. While these proposals may be adequate for small special-purpose systems, they provide no acceptable solution for large information retrieval systems. With the present technology, it is obviously not practical to have a large enough associative memory which can store all of the desired information[1,2] without violating P1. Using small associative memories in conjunction with secondary storage[3] results in considerable amounts of time spent for loading and unloading the associative memory, violating P3.

Somewhat more flexible systems can be obtained by using better data organizations. In the distributed-logic memory,[4,5] data is organized as a single string of symbols divided into substrings of arbitrary lengths by delimiters. Each symbol and its associated control bits are stored in, and processed by, a cell which can communicate with its two neighbors and with a central control unit. In the association-storing processor,[6] the basic unit of data is a triple consisting of an ordered pair of items (each of which may be an elementary item or a triple) and a link which specifies the association between the items. Very complex data structures can be represented conveniently with this method. Even though these two systems provide flexible record formats, they do not satisfy P1.

It is evident that with the present technology, an information retrieval system which satisfies both P1 and P3 is impractical. Hence, trading speed for cost through the use of circulating memory devices seems to provide the only acceptable solution. Delay-line associative devices that have been proposed[7,8] are not suitable for large information retrieval systems because of their fixed word-lengths and small capacities. The use of head-per-track disk or drum memories as the storage medium appears to be very promising because such devices provide a balanced compromise between P1 and P3. An early proposal of this type is the associative file processor[9] which is a highly specialized system. Slotnick[10] points out, in more general terms, the usefulness of logic-per-track devices. Parker[11] specializes Slotnick's ideas and proposes a logic-per-track system for information retrieval applications.

DESIGN PHILOSOPHY OF RAPID

The design of RAPID was motivated by the distributed-logic memory of Lee[4,5] and the logic-per-track device of Slotnick.[10] RAPID provides certain basic pattern matching capabilities which can be combined to obtain more complicated ones. Strings, which are stored on a rotating memory, are read into the cell storage one symbol at a time, processed, and stored back (Figure 1). Processing strings one symbol at a time allows efficient handling of variable-length records and reduces the required hardware for the cells.

Figure 2 shows the organization of data on the rotating memory. Each record is a string of symbols from an alphabet X, which will not be specified here. It is assumed that members of X are represented by binary vectors of length N. Obviously, each symbol must have some control storage associated with it to store the search results temporarily. One control bit has proven to be sufficient for most applications even though some

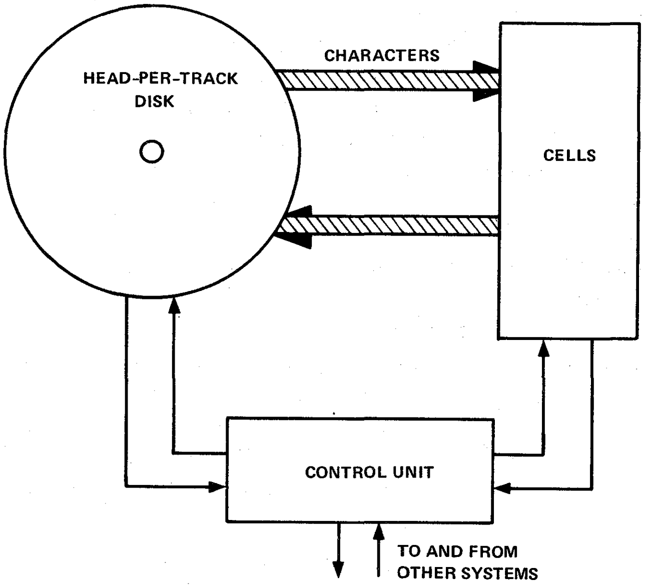


Figure 1—Overall organization of RAPID

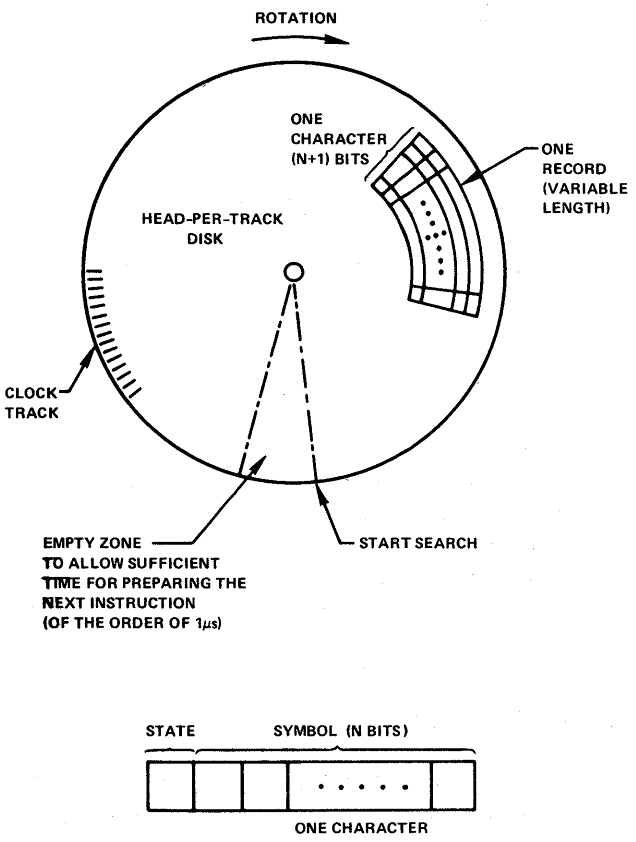


Figure 2—Storage of characters and records

operations may be performed faster with a larger control field. Control information for a symbol will be called its state, $q \in \{0, 1\}$. A symbol x and its state q constitute a character, (q, x).

One of the members of X is a don't-care symbol, δ, which satisfies any search criterion. As an example for the utility of δ, consider an author whose middle name is not known or who does not have one. Then, one can use δ as his middle initial in order to make the author field uniform for all records. We will use the encoding $11 \ldots 1$ for δ in our implementation. In practice, it will become necessary to have other special symbols to delimit records, fields, and so on. The choice of such symbols does not affect the design and is left to the user. It should be emphasized, at this point, that RAPID by itself is only capable of simple pattern matching operations. Appropriate record formats are needed in order to make it useful for a particular information retrieval application. One such format will be given in this paper for general-purpose information retrieval applications.

The idea of associating a state with each symbol is taken from Lee's distributed-logic memory.[4,5] In fact, RAPID is very similar to the distributed-logic memory in principle but differs from it in the following:

1. Only one-way communication exists between neighboring characters in RAPID. This is necessitated because of the use of a cyclic memory but results in little loss in power or flexibility.
2. The use of a cheaper and slower memory makes RAPID more economical but increases the search cycle from microseconds to miliseconds.
3. Besides match for equality, other types of comparisons such as less-than and greater-than are mechanized in RAPID.
4. Basic arithmetic capability is provided in RAPID. It allows for threshold combinations of search functions as well as conventional Boolean combinations.

With the above data organization, the problem of searching for particular sets of records will reduce to that of locating substrings which satisfy certain criteria. Search for successive symbols of a string is performed one symbol per disk or drum revolution. There are at least two reasons for this design choice:

1. At any time, all the cells will be performing identical functions (looking for the same symbol). This reduces the hardware complexity of each cell since the amount of local control is minimized and fewer input and output leads are required.
2. The alternative approach of processing a few symbols at a time fails in the case of overlapping strings. Suppose one tries to process k symbols at a time ($k > 1$) by providing local control for each cell in the form of a counter. Then, if the i-th symbol in the input string is matched, the cell proceeds to match the $(i + 1)$-st symbol. Hence, if one is looking for the pattern ABCA in the string . . .DCABCABCADA. . . , only one of the two patterns will be found. Also, the pattern BCAD will not be found in the above example.

THE CONTROL UNIT

Figure 3 shows a block diagram of RAPID which is a synchronous system operating on the disk clock tracks. The phase signal generator sequences the operations by generating eight phase signals. PHA, PHB, PHC, and PHZ are generated once every disk revolution while PH1, PH2, PH3, and PH4 are generated once every bit time (Figure 4). During PHA, the cell control register (CCR), input symbol register (ISR), and address

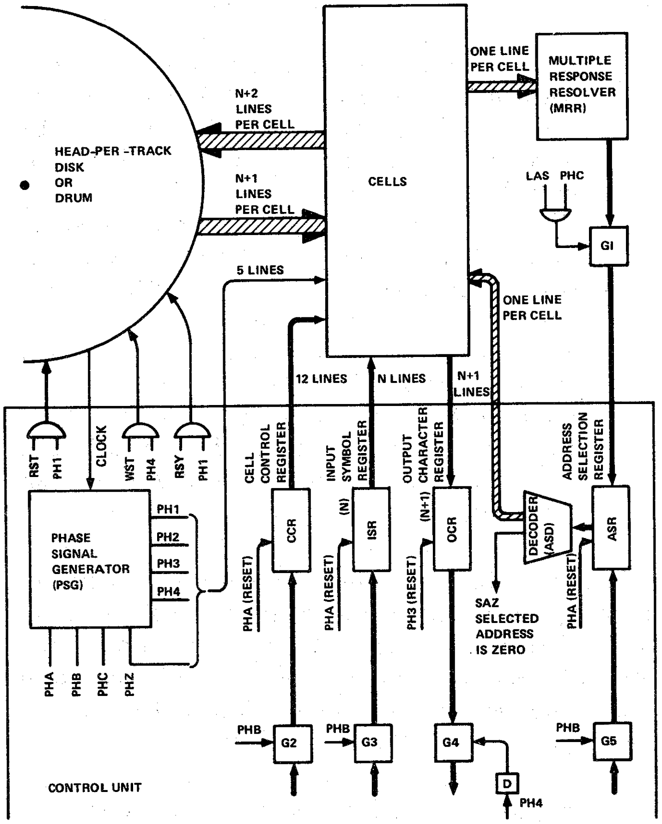


Figure 3—Block diagram of RAPID

selection register (ASR) are cleared. During PHB and PHC, these registers are loaded. Then the execution of the instruction in CCR starts. During PH3, the output character register is reset. It is loaded during PH4 and is unloaded, through G4, after a certain delay.

Most parts of the control unit, namely the instruction sequencing section and the auxiliary registers which are used to load CCR, ISR, and ASR or unload OCR, are not shown in Figure 3. It should be noted, however, that these parts process instructions at the same time that the cells are performing their functions such that the next instruction and its associated data are ready before the next PHB signal. The system can also be controlled by a general-purpose computer which is interrupted during PHB to load the auxiliary registers with the next instruction and associated data.

The arrangement of records on disk is shown in Figure 2. The $N+1$ bits of a character are stored on parallel tracks while the characters of a record are stored serially. One or more clock tracks supply the timing pulses for the system. The empty zone is provided to allow sufficient time for loading the control registers for the next search cycle.

Figure 5 shows the cell control register (CCR) which holds the instruction to be executed for one disk revolution. The function of various fields in this register will now be described.

Read field

This field consists of two bits, RST and RSY. RST commands the cells to read the state bit into the current state flip-flop, CSF. RSY commands the cells to read the symbol bits into the current symbol register, CSR.

Write field

This is similar to the read field and consists of WST and WSY. WST commands that the condition bit, CON (see description of condition field), replace the current state. WSY is a command to replace the current symbol by the contents of current symbol register, CSR, if CON = 1.

Address selection field

This field contains two bits, LAS and RAS. If the LAS bit of this field is set, the address selection register

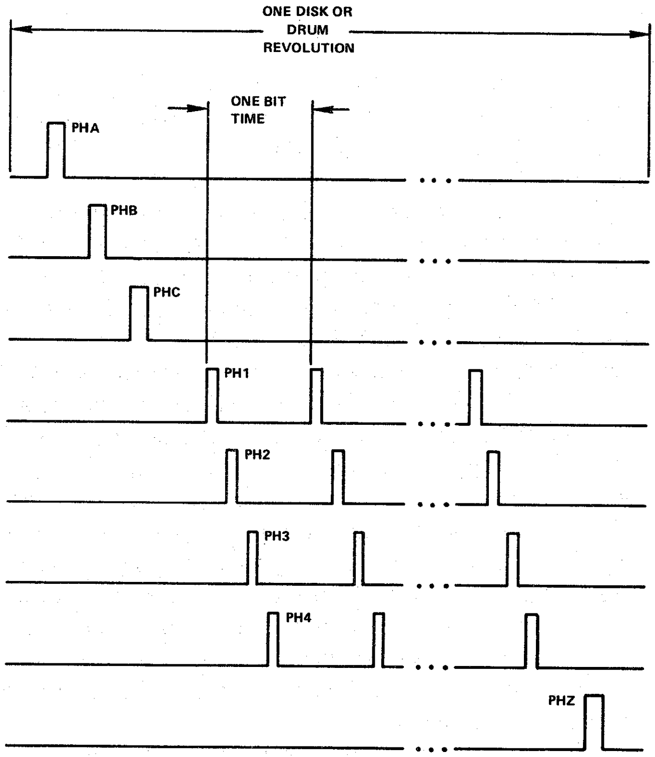


Figure 4—Timing signals

(ASR) is loaded from the multiple response resolver (MRR). MRR outputs the address of the first cell with its ASF on. If the RAS bit is set, the accumulated state flip-flop, ASF, in the cells will be reset. The function of ASF will be described with the cell design. The address selection field allows the sequential readout of the tracks which contain information pertinent to a search request.

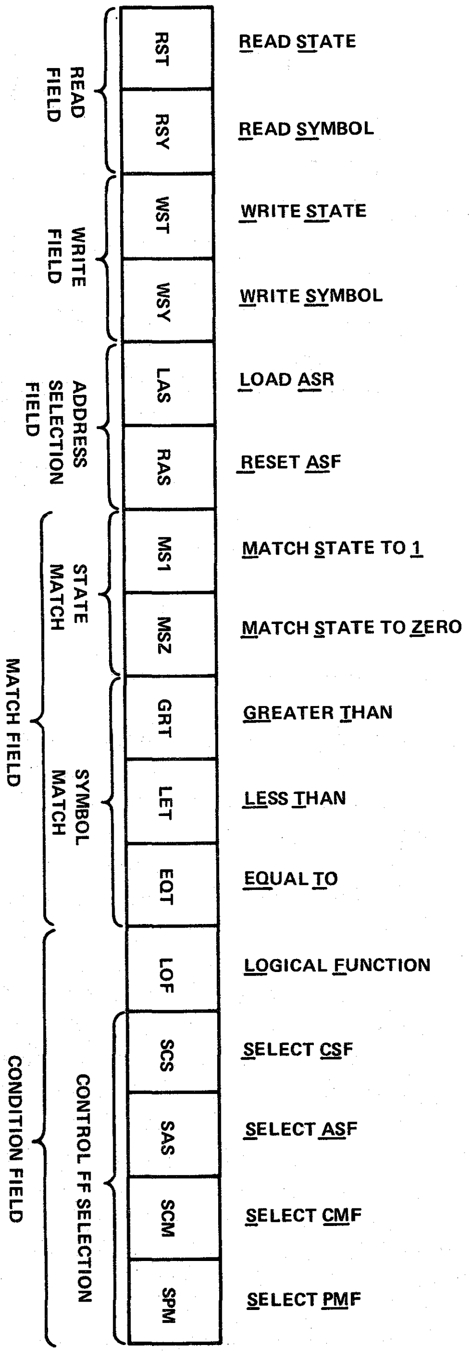


Figure 5—The cell control register (CCR)

TABLE I—The Match Condition for the State Part of a Character

MS1	MSZ	Match
0	0	never
0	1	if $q = 0$
1	0	if $q = 1$
1	1	always

Match field

This field consists of two subfields; the state match subfield, and the symbol match subfield. These subfields specify the conditions that the state and symbol of a character must meet. If both conditions are satisfied for a particular character, the current match flip-flop (CMF) of the corresponding cell is set. The state match subfield consists of MS1 and MSZ. The conditions for all combinations of these two bits are given in Table I. The symbol match subfield consists of three bits; GRT, LET, and EQT. All the symbols in the cells are simultaneously compared to the 1's complement of the contents of ISR. Table II gives the conditions for all combinations of the three signals. S is the symbol in a cell and $\bar{Y}$ is the 1's complement of the contents of ISR.

Condition field

This field specifies how the condition bit, CON, is to be computed from the contents of the following four flip-flops in a cell: current state flip-flop, CSF; accumulated state flip-flop, ASF; current match flip-flop, CMF; and previous match flip-flop, PMF. LOF specifies the logical function to be performed (AND if LOF=1, OR if LOF=0). The other four bits in this field specify a subset W of the set of four control flip-flops on which the logical function is to be performed. For example, if SCS=1, then CSF $\in$ W.

TABLE II—The Match Condition for the Symbol Part of a Character

GRT	LET	EQT	Match
0	0	0	never
0	0	1	if $S = \bar{Y}$ or $S = \delta$
0	1	0	if $S < \bar{Y}$ or $S = \delta$
0	1	1	if $S \leq \bar{Y}$ or $S = \delta$
1	0	0	if $S > \bar{Y}$ or $S = \delta$
1	0	1	if $S \geq \bar{Y}$ or $S = \delta$
1	1	0	if $S \neq \bar{Y}$ or $S = \delta$
1	1	1	always

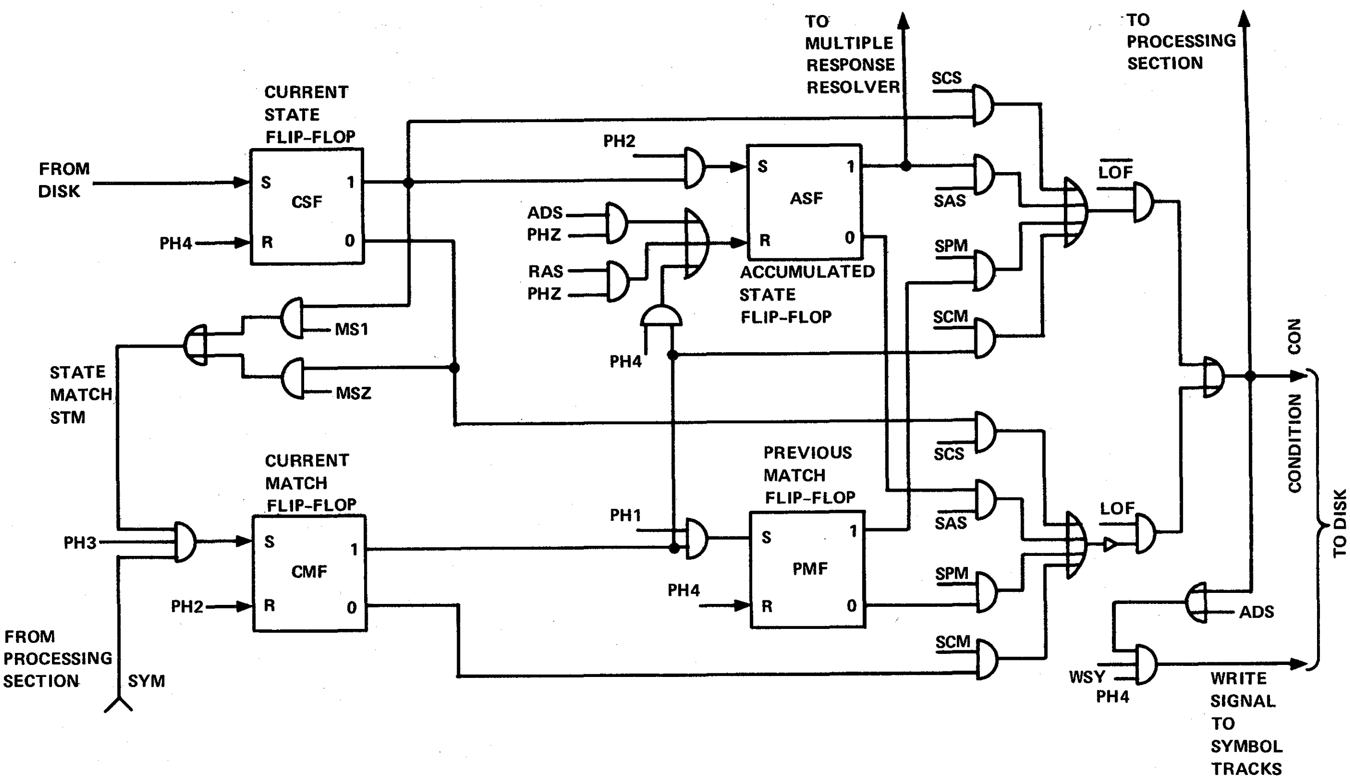


Figure 6—Control section of a cell

As will be seen later, the cell design is such that by appropriate combinations of bits in CCR, other functions besides simple comparison can be performed.

THE CELL DESIGN

Each cell consists of two sections; the control section, and the processing section. Roughly speaking, the control section processes the state part of a character while the processing section operates on the symbol part.

The control section (Figure 6) contains four flip-flops: current state flip-flop, CSF; accumulated state flip-flop, ASF; current match flip-flop, CMF; and previous match flip-flop, PMF. CSF contains the state of the character read most recently from the disk. ASF contains the logical OR of the states of characters read since it was reset. This flip-flop serves two purposes: finding out which tracks contain at least one character with a set state (reset by ADS during PHZ) and propagating the state information until a specified character is encountered (reset by RAS during PHZ and by CMF during PH4). CMF contains (after PH3) the result of current match. It is set if both the state and symbol of the current character meet the match specifications. Finally, PMF contains the match result for the previous character.

The condition signal, CON, is a logical function of the contents of control flip-flops. The four signals SCS, SAS, SCM, and SPM select a subset of these flip-flops and the logical function signal, LOF, indicates whether the contents of selected flip-flops should be ANDed (LOF=1) or ORed (LOF=0) together to form CON. The value of CON will replace the state of current character if the write state signal, WST, is activated.

The address selection signal, ADS, is activated by the address selection decoder. This signal allows conventional read and write operations to be performed on selected tracks of the disk. It is also possible, through the multiple response resolver, to read out sequentially the contents of tracks whose corresponding ASF's are set.

The processing section, shown in Figure 7, contains an N-bit adder with inputs from ISR and the current symbol register, CSR. During PH1, a symbol is read into CSR. During PH2, contents of CSR are added to contents of ISR with the result stored back in CSR. Overflow indication is stored in the overflow flip-flop, OFF. Before the addition takes place, the don't-care flip-flop, DCF, is set if CSR contains the special don't-care symbol δ. From the results of addition, it is decided whether the symbol satisfies the search specification (SYM = 1 if it does, SYM = 0 if it does not).

The adder in each cell allows us to add the contents of ISR to the current symbol or to compare the symbol to the 1's complement of the contents of ISR. If we denote the current symbol by S, the contents of ISR by Y, and its 1's complement by $\bar{Y}$, then:

$$S=\bar{Y} \text{ iff } S+Y+1=2^N$$

$$S>\bar{Y} \text{ iff } S+Y+1>2^N$$

$$S<\bar{Y} \text{ iff } S+Y+1<2^N$$

N is the length of the binary vector representation of S and Y. Hence if we denote the result of addition in CSR by Z and the overflow by OFF, we have:

$$S=\bar{Y} \text{ iff } Z=0$$

$$S>\bar{Y} \text{ iff } Z\neq 0 \text{ and OFF}=1$$

$$S<\bar{Y} \text{ iff OFF}=0$$

Note that the carry signal into the adder is activated if any one of the signals GRT, LET, or EQT is active. The above equations are used in the design of the circuit which computes the symbol match result, SYM (upper right corner of Figure 7). The result of symbol match is ANDed with the result of state match (STM) during PH3 to set the current match flip-flop.

Finally, during PH4, the contents of CSR can be written onto the disk or put on the output bus. Since the address selection line, ADS, is active for at most one cell, no conflict on the output bus will arise.

EXAMPLES OF APPLICATIONS

We first give a set of 12 instructions for RAPID. These instructions perform tasks that have been found to be useful in information retrieval applications. Each instruction, when executed by RAPID, will load CCR with a sequence of patterns. These sequences of patterns are also given. We restrict our attention to search

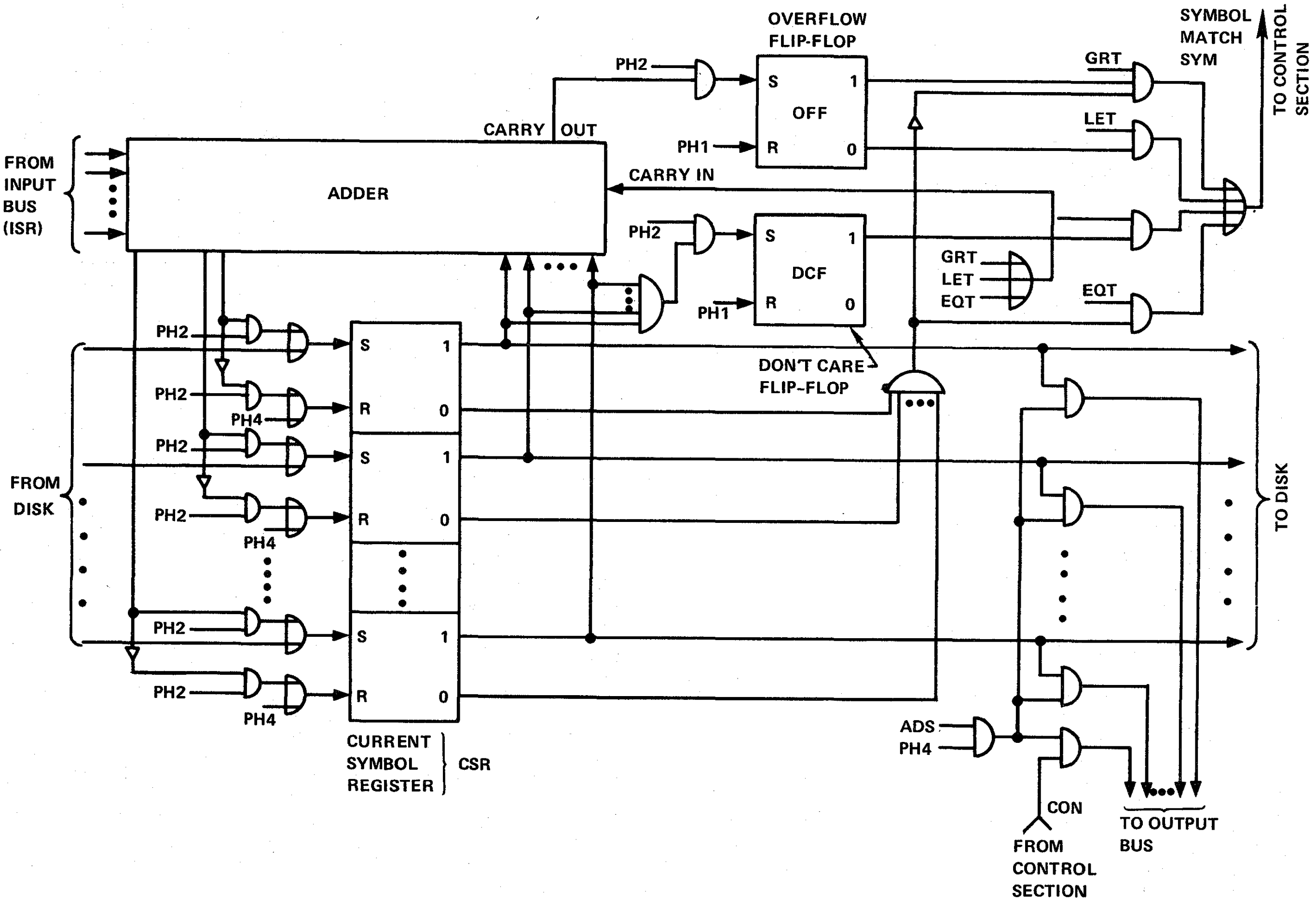


Figure 7—Processing section of a cell

instructions only. Input and output instructions must also be provided to complete the set.

1. **search and set** s: Find all occurrences of the symbol s and set their states.
2. **search for** $s_1s_2 \ldots s_n$: Find all the occurrences of the string $s_1s_2 \ldots s_n$ and set the state of the symbols which immediately follow s_n.
3. **search for marked** $s_1s_2 \ldots s_n$: Same as the previous instruction except that for a string to qualify, the state of its first symbol must be set.
4. **search for marked** ψ s: Search for symbols whose states are set and have the relation ψ with s. Then, set the state of the following symbol. Possible relations are $<$, $\leq$, $>$, $\geq$, and $\neq$.
5. **propagate to** s: If the state of a symbol is set, reset it and set the state of the first s following it.
6. **propagate** i: If the state of a symbol is set, reset it and set the state of the i-th symbol to its right.
7. **expand to** s: If the state of a symbol is set, set the state of all symbols following it up to and including the first occurrence of s.
8. **expand** i: If the state of a symbol is set, set the state of the first i symbols following it.
9. **contract** i: If the state of a symbol is reset, reset the state of the first i symbols following it.
10. **expand** i **or to** s: If the state of a symbol is set, perform 7 if an s appears within the next i symbols; otherwise, perform 8.
11. **add** s: Add the numerical value of s to the numerical value of any symbol whose state is set.
12. **replace by** s: If the state of a symbol is set, replace the symbol by s.

The microprograms for these instructions are given

TABLE III—Microprograms for RAPID Instructions

Number	Instruction		Repetition	Contents of ISR	Contents of CCR															
					Read Field		Write Field		Address Selection		Match Field					Condition Field				
											State		Symbol			Logic	FF Selection			
					RST	RSY	WST	WSY	LAS	RAS	MS1	MSZ	GRT	LET	EQT	LOF	SCS	SAS	SCM	SPM
1	search and set s		1	$\bar{s}$		1	1	0	0		1	1	0	0	1		0	0	1	0
2	search for $s_1s_2 \ldots s_n$		1	$\bar{s}_1$		1	1	0	0		1	1	0	0	1		0	0	0	1
			j=2 to n	$\bar{s}_j$	1	1	1	0	0		1	0	0	0	1		0	0	0	1
3	search for marked $s_1s_2 \ldots s_n$		j=1 to n	$\bar{s}_j$	1	1	1	0	0		1	0	0	0	1		0	0	0	1
4	search for marked ψs	$<$	1	$\bar{s}$	1	1	1	0	0		1	0	0	1	0		0	0	0	1
		$\leq$	1	$\bar{s}$	1	1	1	0	0		1	0	0	1	1		0	0	0	1
		$>$	1	$\bar{s}$	1	1	1	0	0		1	0	1	0	0		0	0	0	1
		$\geq$	1	$\bar{s}$	1	1	1	0	0		1	0	1	0	1		0	0	0	1
		$\neq$	1	$\bar{s}$	1	1	1	0	0		1	0	1	1	0		0	0	0	1
5	propagate to s		1	$\bar{s}$	1	1	1	0	0	1	1	1	0	0	1	1	0	1	1	0
6	propagate i		i		1		1	0	0		1	0	1	1	1		0	0	0	1
7	expand to s		1	$\bar{s}$	1	1	1	0	0	1	1	1	0	0	1		0	1	0	0
8	expand i		i		1		1	0	0		1	0	1	1	1	0	1	0	0	1
9	contract i		i		1		1	0	0		1	0	1	1	1	1	1	0	0	1
10	expand i or to s		i	$\bar{s}$	1	1	1	0	0		1	1	1	1	0	1	1	0	1	0
					1		1	0	0		1	0	1	1	1	0	1	0	0	1
11	add s		1	s	1	1		1	0								1	0	0	0
12	replace by s		1	s	1	0		1	0								1	0	0	0

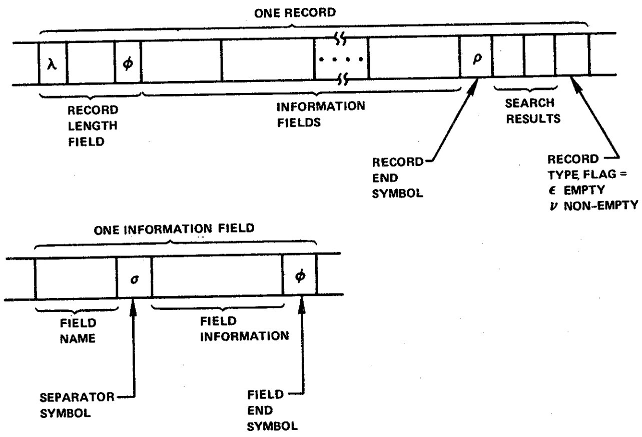


Figure 8—Data storage format

in Table III. A blank entry in this table constitutes a don't-care condition. The entries in the repetition column specify the number of times the given patterns should be repeated. As can be seen from Table III, this set of instructions does not exploit all the capabilities of RAPID since some of the bits in the CCR assume only one value (0 or 1) for all the instructions.

To illustrate the applications of RAPID, we first choose a format for the records (Figure 8). The record length field must have a fixed length in order to allow symbol by symbol comparison of the record length to a given number. The information fields can be of arbitrary lengths. The flag field contains three characters; two for holding the results of searches, and one which contains a record type flag. The Greek letters used on Figure 8 are reserved symbols and should not be used except for the purposes given in Table IV.

As mentioned earlier, a special symbol, δ, is used as a don't-care symbol. It is also helpful to have a reserved symbol, τ, which can be used as temporary substitute for other symbols during a search operation. Let us now consider two simple examples to show the utility of the given instruction set.

Example 1. Assuming that the record length is specified by one symbol, the following program marks all the empty records whose lengths are not less than s. This is useful when entering a new record of length s to find which tracks contain empty records that are large enough.

```
search for $\lambda$
search for marked $\geq s$
propagate to $\rho$
propagate 3
search for marked $\epsilon$
```

If the record length is specified by two characters, we note that $t_1t_2 \geq s_1s_2$ iff $t_1 > s_1$ or $t_1 = s_1$ and $t_2 \geq s_2$. Hence, we write the following program:

```
search for $\lambda$
search for marked $> s_1$
propagate 1
replace by $\tau$
search for $\lambda$
search for marked $s_1$
search for marked $\geq s_2$
replace by $\tau$
search and set $\tau$
replace by $\phi$
propagate to $\rho$
propagate 3
search for marked $\epsilon$
```

Example 2. The following program marks all non-empty records which contain in their title field, designated by TI, a word having "magnet" as its first six characters and having 3 to 10 non-blank characters after that. β designates the "blank" character.

```
search for $\phi$TI$\sigma$
expand to $\phi$
search for marked magnet
expand 10 or to $\beta$
contract 3
propagate to $\rho$
propagate 3
search for marked $\nu$
```

It is important to note that the record format given here serves only as illustration. Because of its generality and flexibility, this format is not very efficient in terms of storage overhead and processing speed. For any given application, one can probably design a format which is more efficient for the types of queries involved.

CONCLUSION

In this paper, we have described a special-purpose highly parallel system for information retrieval applica-

TABLE IV—List of Reserved Symbols

λ	Indicates start of *length* field.
ρ	Indicates end of a *record.*
σ	*Separates* name and information subfields in a field.
ϕ	Indicates end of a *field.*
ϵ	Designates the end of an *empty* record.
ν	Designates the end of a *non-empty* record.
δ	Is the *don't-care* symbol.
τ	Is used as *temporary* substitute for other symbols.

tions. This system must be evaluated with respect to the properties of an ideal information retrieval system summarized earlier. It is apparent that RAPID satisfies P2, P4 and P5. The extent to which P1 and P3 are satisfied by RAPID is difficult to estimate at the present.

With respect to P1, the storage medium used has a low cost per bit. However, the cost for cells must also be considered. Because of the large number of identical cells required, economical implementation with LSI is possible. Figures 6 and 7 show that each cell has one N-bit adder, $N+6$ flip-flops, $6N+39$ gates, and $4N+23$ input and output pins. For a symbol length of $N=8$ bits, each cell will require no more than 250 gates and 60 input and output pins. The number of input and output pins can be reduced considerably at the expense of more sophisticated gating circuits (i.e., sharing input and output connections).

With respect to P3, the search speed depends on the number of symbols matched. If we assume that on the average 50 symbols are matched, the matching phase will take about 70 disk revolutions (to allow for overhead such as propagation of state information and performance of logical operations on the search results). Hence, the search time for marking the tracks which contain relevant information is of the order of a few seconds.

Some important considerations such as input and output of data and fault-tolerance in RAPID have not been explored in detail and constitute possible areas for future research. The interested reader may consult Reference 12 for some thoughts on these topics.

ACKNOWLEDGMENTS

The author gratefully acknowledges the guidance and encouragement given by Dr. W. W. Chu in the course of this study. Thanks are also due to Messrs. P. Chang, D. Patterson, and R. Weeks for stimulating discussions.

REFERENCES

1 J GOLDBERG M W GREEN
Large files for information retrieval based on simultaneous interrogation of all items
Large-capacity Memory Techniques for Computing Systems New York Macmillan pp 63-67 1962

2 S S YAU C C YANG
A cryogenic associative memory system for information retrieval
Proceedings of the National Electronics Conference pp 764-769 October 1966

3 J A DUGAN R S GREEN J MINKER W E SHINDLE
A study of the utility of associative memory processors
Proceedings of the ACM National Conference pp 347-360 August 1966

4 C Y LEE
Intercommunicating cells, basis for a distributed-logic computer
Proceedings of the FJCC pp 130-136 1962

5 C Y LEE M C PAULL
A content-addressable distributed-logic memory with applications to information retrieval
Proceedings of the IEEE Vol 51 pp 924-932 June 1963

6 D A SAVITT H H LOVE R E TROOP
ASP; a new concept in language and machine organization
Proceedings of the SJCC pp 87-102 1967

7 W A CROFUT M R SOTTILE
Design techniques of a delay line content-addressed memory
IEEE Transactions on Electronic Computers Vol EC-15 pp 529-534 August 1966

8 P T RUX
A glass delay line content-addressable memory system
IEEE Transactions on Computers Vol C-18 pp 512-520 June 1969

9 R H FULLER R M BIRD R M WORTHY
Study of associative processing techniques
Defense Documentation Center AD-621516 August 1965

10 D L SLOTNICK
Logic per track devices
Advances in Computers Vol 10 pp 291-296 New York Academic Press 1970

11 J L PARKER
A logic-per-track retrieval system
Proceedings of the IFIPS Conference pp TA-4-146 to TA-4-150 1971

12 B PARHAMI
RAPID; a rotating associative processor for information dissemination
Technical Report UCLA-ENG-7213 University of California at Los Angeles February 1972

The architecture of a context addressed segment-sequential storage

by LEONARD D. HEALY

U.S. Naval Training Equipment Center
Orlando, Florida

and

GERALD J. LIPOVSKI and KEITH L. DOTY

University of Florida
Gainesville, Florida

INTRODUCTION

This paper presents a new approach to the problem of searching large data bases. It describes an architecture in which a cellular structure is adapted to the use of sequential-access bulk storage. This organization combines most of the advantages of a distributed processor with that of inexpensive bulk storage.

Large data bases are required in information retrieval, artificial intelligence, management information systems, military and corporate logistics, medical diagnosis, government offices and software systems for monitoring and analyzing weather, ecological and social problems. In fact, most nonnumerical processing requires the manipulation of sizable data bases. An examination of memory costs indicates that at present the best way of storing such data bases, and the one most widely used in new computer systems, is disc storage. However, the disc is not used anywhere near its full potential.

Discs are presently used as random access storages. Each word has an address which is used to select the word. However, the association of each word with a fixed location, required in a random access storage, is a disadvantage. In a fixed-head disc, each word is read by means of a read head and can be over-written by a write head. Now, if we discard the capability to randomly address, associative addressing can be used as words are read, and automatic garbage collection can be performed as words are rewritten.

Perhaps the most important feature of this architecture is its associative (or context) addressing capability. Search instructions are used to mark words in storage that match the specified criteria. Context addressing is achieved by making the search criteria depend upon both the content of the word being searched and the result of previous searches. For example, consider the search of a telephone directory in which each entry consists of three separate, contiguously placed words: subscriber name, subscriber address, and telephone number. The search for all subscribers named John J. Smith is a content search—a search based upon the content of a single word. The search for all subscribers named Smith who live on Elm Street is a context search—the result of the search for one word affects the search for another.

Associative addressing, or more correctly, content addressing, has been attempted on discs[1] in which each word in the memory is a completely separate entity in such an addressing scheme. This paper shows how context addressing can be done. Words nearby a word in the storage can be searched in context, such that a successful search for one word can be made dependent on a history of successful searches on the nearby words. Strings, sets, and trees can be stored and searched in context using such a context-addressed storage.[2] More complex structures such as relational graphs can also be efficiently searched.

The context-addressed disc has the following advantage over a random-accessed disc in most nonnumeric data processes. Large data bases can be searched, for instance, for a given string of characters. Once a string is found, data stored nearby the string on the disc track can be returned to the central processor. Only relevant data need be returned, because the irrelevant data can be screened out by context-addressed searching on the disc itself to select the relevant data. In contrast, a conventional disc will return considerable irrelevant data to the central processor to be searched. Thus, the I/O channel requirements and primary storage requirements of the computer are reduced because less data is transferred. In fact, there is a maximum number of random-accessed discs that can be serviced by a central processor because it has to search through all the irrelevant data returned by all the discs, whereas an unlimited number of context-addressed discs can be searched in parallel. Moreover, the instructions used to search the disc storage can be stored in the disc storage itself. Thus, the central processor can transfer a search program to the disc system, then run independently until the disc has found the data. The computer would be interrupted when the data was found. This will reduce the interrupt load on the computer.

In this paper we therefore study the implementation of a context-addressed storage using a large number of discs. The segment-sequential storage to be studied will have the following characteristics (see Figure 1). The entire storage will store a 1-dimensional array of words, called the file. From the software viewpoint, collections of words related in a data structure format are stored in a contiguous section of the file, called a record. Records can be of mixed size. From the hardware viewpoint, the file will be broken into equal-length segments and stored on fixed-head discs, one segment to a disc. In the time taken to rotate one disc completely, all discs can search simultaneously for a given word in the context of a data structure as directed by the user's query, marking all words satisfying the search. Words selected by such context searches can be over-written with new data in such a data structure, erased, read out to the I/O channel, or selected as instructions to be executed during the next disc rotation. Data in groups of words can be copied and moved from one part of the file to another part as the data structure is edited. In the meantime, a hardware garbage collection algorithm will collect erased words to the bottom of the file so that large aggregates of words are available to receive large records.

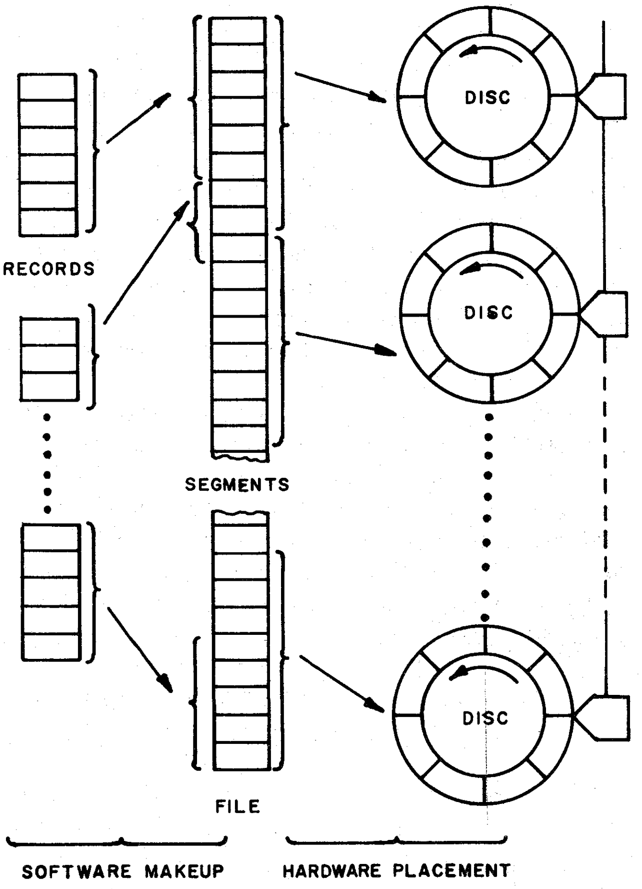


Figure 1—Storage of records as segments

MOTIVATION

The problem that leads to the system architecture proposed here is the efficient use of storage devices equivalent to large disc storages. Access to files stored on such devices is currently based upon a sequential search of the file area by reading blocks of data into the main storage of the central processor and searching it there or by use of a file index which somehow relates the file content to its physical location. Many hierarchies of searches have been devised—all efforts to solve the basic problem that the storage device is addressed by location but the data is addressed by its content.

The advantage of information retrieval based upon content is well documented.[3,4,5] However, the trend has been toward application of associative-search hardware within the central computer. Content-search storages have been implemented as subsystems within a computer system;[6,7] but even in these cases, the use of the search subsystem has been closely associated with operations in the central processor. The devices fit into the storage hierarchy between the central processor and the main core storage. A typical application of a content-addressed storage is as a cross-reference to information in main storage—the cache storage. An associative storage subsystem specifically designed for the processing useful in software applications has been proposed,[8] but even that is limited in size by the cost of the special storage hardware.

Systems of the type mentioned are small, high-speed units. They are limited to content search and are restricted in size relative to bulk storage devices. Their application to searching of large data bases is limited to general improvement of central processor efficiency or to searching the index for a large data base. What is needed for true context search of a large data base is an economic subsystem which can be connected to a computer and can perform context search and retrieval operations on a large data base stored within that subsystem.

The approach described in this paper provides just such a subsystem. It is a semi-autonomous external device which has its own storage and control logic. The design concept is specifically oriented toward use of a large bulk storage medium instead of high-speed core storage. In addition, the processing capability of the subsystem has been expanded to include not only list processing, but also special searches such as matching data strings against templates and operations on bit strings to simulate networks of linear threshold elements useful in pattern recognition.

The basic building block of the proposed architecture is a segmented sequential storage. The sequential storage was chosen because it provides an economically feasible way to store a large data base. In order to perform search operations on this data base, the storage must be divided into segments which can be searched in parallel. Each segment of the sequential storage must have its own processing capability for conducting such a search. This leads to a cellular organization in which each cell consists of a sequential storage segment.

The segment-sequential storage has the following property. Suppose n items are compared with each other exhaustively. This requires n storage words. Thus, the total size of the storage obviously grows linearly with n. However, as the size grows, more discs are added on, but the time for a search depends only on the size of the largest disc and not on the number of discs. Thus, the time to search for each item in a query is still the same. The total time for the search grows linearly with the number of words to be compared. As a first approximation to the cost of programming, the product of storage size and search time grows as n^2. This compares with n^3 for a conventional computer. Thus, this storage is very useful for those operations in which all the elements in one set are exhaustively compared with each other or with members of another set, especially when the set is very large. Similarly, the cost of a comparison of one element with a set of n elements grows as n^2 in a conventional processor, and as n in this architecture. The rate of growth of the cost of programming for this architecture is the same as for cellular associative memories,[9] primarily because it too is a parallel cellular system.

Some algorithms demand exhaustive comparisons. Some of these are not used because of their extreme cost. Other algorithms abandon exhaustive comparison to be usable in the Von Neumann computer at some increase in programming complexity, loss of relevance or accuracy, or at the expense of structuring the data base so that other types of searches cannot be efficiently conducted. In view of the lower cost of an exhaustive search, this storage might be useful for a number of algorithms which are now used for information management in the Von Neumann computer and many others which are not practical on that type of machine.

Discs appear to be slow, but their effective rate of operation can be made very fast when they are used in parallel. A typical disc rotates at sixty revolutions per second. The segment-sequential storage will be able to execute sixty instructions per second. (Faster rates may eventually be possible with special discs, or on processors built from magnetic bubble memories, semiconductor shift registers, or similar sequential memories.) However, if one hundred fixed-head discs storing 32k words per disc are simultaneously searched, nearly two hundred million comparisons per second are performed. This is approximately the rate of the fastest processor built. This large system of 100 discs would cost about $5000 per disc for a total cost of $500,000. This cost is small compared to that of a new large computer. Thus, this architecture appears to be cost-effective.

This architecture is based on storage and retrieval from a segmented sequential table data structure utilizing associative addressing. This results in the following characteristics.

(1) The search time is independent of the file size. The data content of each cell is searched in parallel; the search time depends only upon the cycle time of the individual storage segment and the number of instructions in the query.

(2) The search technique is based largely upon context. No tables or cross references are required to locate data. However, there are cases where cross references can be used to advantage.

(3) New data may be inserted at any place in the file. The moving of the data that follows the place of insertion to make room for the new information is performed automatically by the cells.

(4) Whenever information is deleted from the file, later file entries will be moved to close the gap. Thus, the locations in the bulk storage will always be "packed" to put available storage at the end of the file area.

(5) The system is a programmable processor. Since

each instruction takes 1/60 second to be executed, as much processing should be done as possible for each instruction. Further, because the cell is large, the cost of the processing hardware will be amortized over many words in that cell. Thus, a large variety of rather sophisticated instructions will be used to search and edit the data. Programming with these instructions will be simpler than programming a conventional computer in assembler language.

Lastly, since this architecture is basically cellular, where one disc and assoçiated control hardware is a (large) cell, the following advantages can be obtained.

(1) The system is iterative. The design of one cell is repeated. The cost of design is therefore amortized over many cells.
(2) The system is upward expandable. An initial system can be built with a small number of cells. As storage demands increase, more cells can be added. The old system does not have to be discarded.
(3) The system is fail soft. If a cell is found to be faulty, it can be taken out of the system, which can still operate with reduced capability.
(4) The system is restructurable. If several small data bases are used, the larger system can be electrically partitioned so that each block of cells stores and searches one data base independently of the other blocks. Further, several systems attached to different computers, say in a computer network, can be tied together to make one larger system. Since the basic instruction rate is only sixty instructions per second, the time delays of data transmission through the network are generally insignificant. Thus, the larger system can operate as fast as any of its cells for most operations.

Based on these general observations, the segment-sequential storage has very promising capabilities. In the next sections, we will describe the machine organization and show some types of problems that are easily handled by this system.

SYSTEM ORGANIZATION

The system block diagram for the segment-sequential storage is shown in Figure 2. The system consists of a controller plus a number of identical cells. The controller provides the interface with an I/O channel of the central computer necessary to perform: (1) input and output

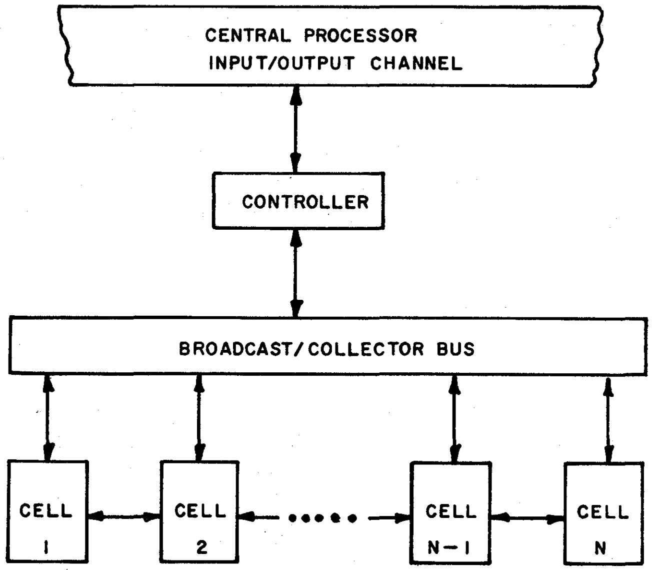


Figure 2—System block diagram.

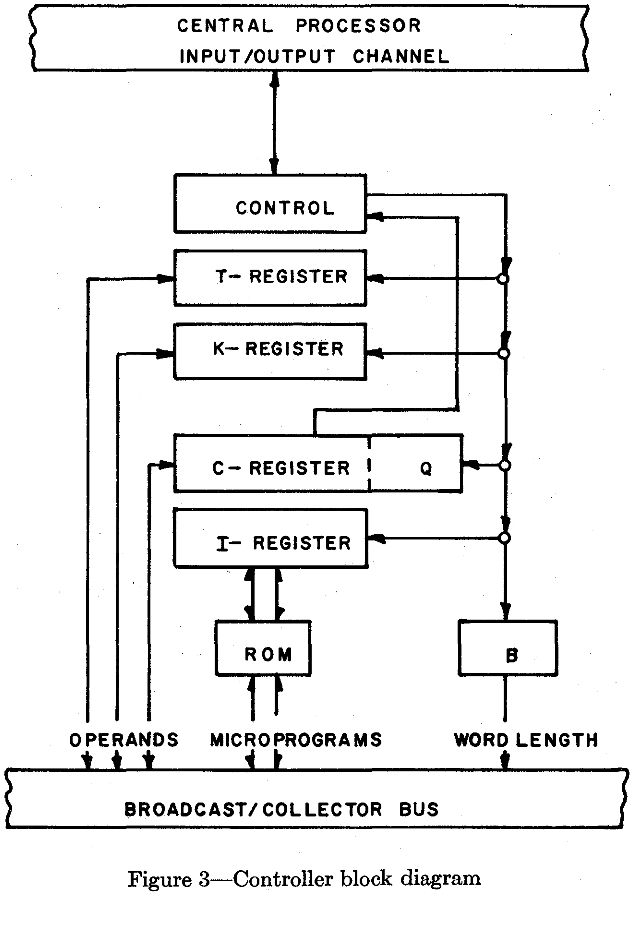


Figure 3—Controller block diagram

operations between the central computer's core storage and the storage of the individual cells, and (2) search operations commanded by the central computer. Each individual cell communicates with the controller via the broadcast/collector bus and with its left and right adjacent neighbor by a direct connection. All cells are identical in structure.

A more detailed diagram of the controller is shown in Figure 3. The controller appears similar to a conventional disc controller to the central computer. It performs the functions necessary to execute orders transmitted from the central computer via its I/O channel. The segment-sequential storage is thus able to perform its specialized search operations under the command of a program in the I/O channel. Intervention of the central computer is required only for initiation of a search and, perhaps, for servicing an interrupt when the search is complete.

In its role in providing the interface between the I/O channel and the cells, the controller is quite different from a conventional disc controller. Instead of registers for track and head selection, this controller provides the registers required to hold the information needed by the cells in performing their specialized search operations. These registers are:

(1) Instruction Register—I: This register holds the instruction which tells what function the cells should perform during the next cycle. The instruction is decoded by a read-only memory that breaks it down into microinstructions.
(2) Comparand Register—C: This register holds the bit configuration representing the character being searched for. It has an extension field Q which is used when writing data into the cell storage.
(3) Mask Register—K: This register holds a mask which specifies which bits of the C Register are to be considered in the search.
(4) Threshold Register—T: This register holds a threshold value which allows use of search criteria other than exact match or arithmetic inequality.
(5) Bit-length Register—B: This register is used to hold the number of bits in the data word. This allows the word size of the storage segments to be selected under control of the computer.

A block diagram of the cell is shown in Figure 4. Each cell executes the commands broadcast by the controller and indicates the results by transmission of information to the broadcast/collector bus and also through separate signal lines to its adjacent neighbors. The C, K, T, and B Registers of the controller are duplicated in each cell. These registers are used by the arithmetic unit in each cell in performing the commanded operation upon its segment of the storage. The status register is used to hold composite information about the flag bits associated with individual words in the storage segment. Control logic in the cell determines what signals are passed from the cell to the broadcast/collector bus and to adjacent cells. Each cell can transfer its entire storage contents to its neighbor.

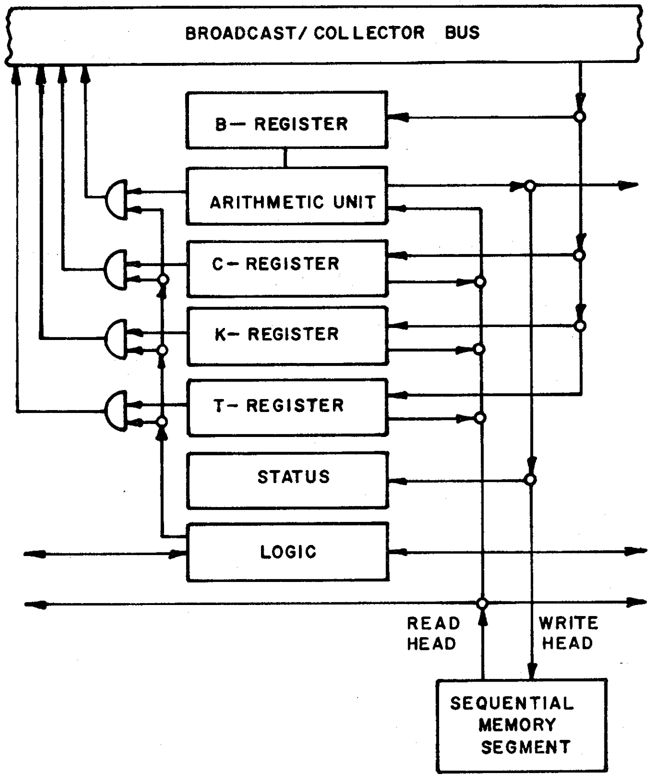


Figure 4—Block diagram of cell

DATA FORMAT

The storage structure of the segment-sequential storage system consists of a number of cells, each of which contains a fixed-length segment of the total sequential storage. Figure 5 depicts the arrangement of data words within one such segment. The storage segment within the cell is a sequential storage device such as a track on a drum or disc, a semiconductor shift register, or a magnetic bubble storage device. Words stored in the segment are stored sequentially, beginning at some predefined origin point. Data read at the read head is appropriately processed by the arithmetic unit and written by the write head.

The information structure of the segment-sequential storage system consists of fixed-length words arranged

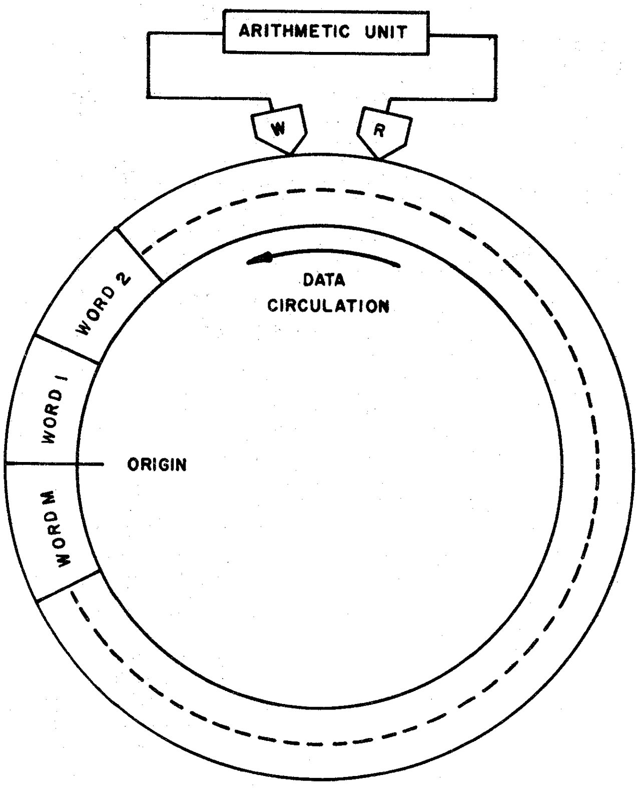


Figure 5—Word arrangement in a storage segment

in variable-length records. The words in a record are stored in consecutive storage locations (where the location following the last storage location in a segment is the first storage location in the following segment). Thus, a record may occupy only a part of one storage segment or occupy several adjacent segments. The start of a record is indicated by a flag bit attached to the first word in the record, and an end of a record is implied by the start of the next record. Figure 6 shows how a record may be spread over several adjacent segments.

Figure 7 shows an expanded view of one word in storage. The b data bits in the word are arranged serially, least significant bit first, with four flag bits terminating the word. The functions of the flag bits are:

(1) S: The *S*TART bit is used to indicate the beginning of a data set (record). The search of a record begins with a word containing a START bit.
(2) P: The *P*ERMANENT bit is used for special designations. Interpretation of this bit depends upon the instruction being executed by the cell.
(3) M: The *M*ATCH bit is used to mark words which satisfy the search criteria at each step in the context search operations.

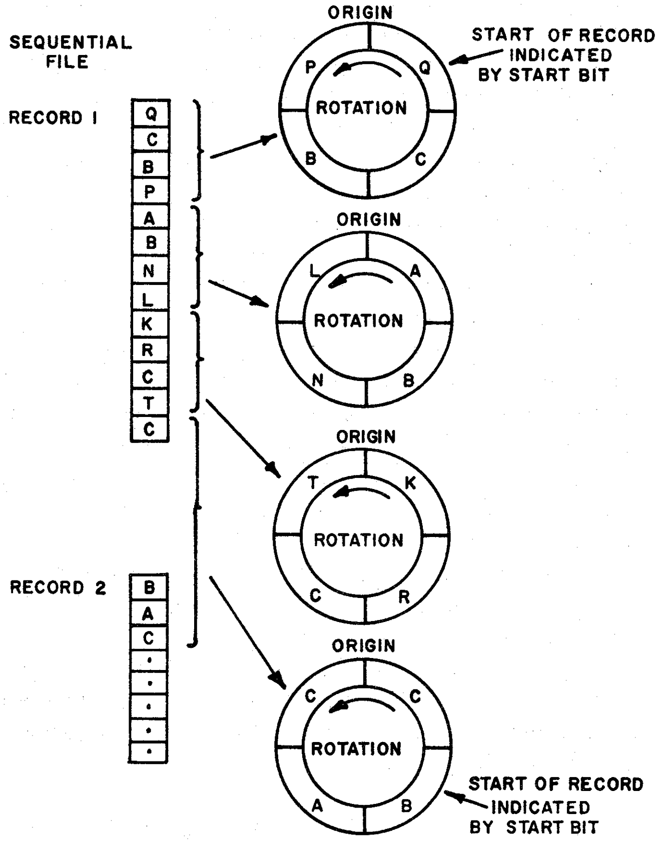


Figure 6—Division of a file into fixed-length segments

(4) X: The *X* bit is used to mark deleted words. Words so marked are ignored and are eventually overlaid in an automatic storage compression scheme.

OPERATIONAL CONCEPTS

The basic operation in context searching is a search for records which satisfy a criterion dependent upon both content and the result of previous searches. As an

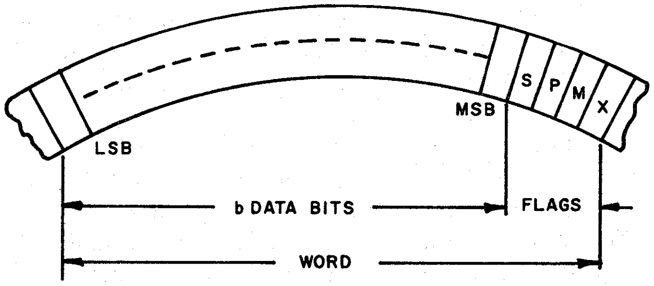


Figure 7—Word configuration

example to illustrate how the segment-sequential storage is able to search all cells simultaneously, consider the ordered search for the characters A, B, C. That is, determine which records contain A, B, and C in that order but not necessarily contiguous.

The three searches required to mark all records that satisfy such a query are:

(1) Mark all words in storage which contain the character A.
(2) Mark all words in storage which contain the character B and follow (not necessarily immediately) a previously marked character in the same record. At the same time, reset the match indication from the previous search.
(3) Repeat the operation of step 2 for the character C.

The result of these steps is to leave marked only those records which match the ordered search specified.

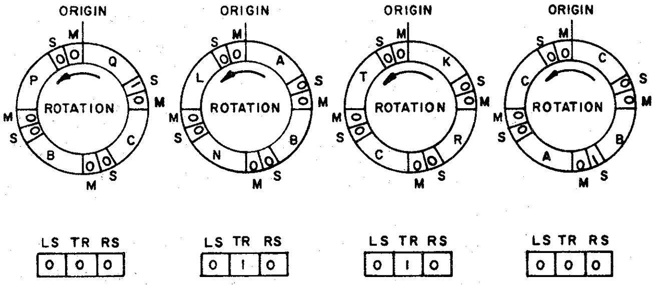


Figure 8a—Flag and status bits before start of search

Figure 8 shows four segments of a system which will be used to illustrate the processing of such a search. The storage segments each contain four words (characters). Only the START and MATCH flags are indicated.

The origin (beginning) of each segment is at the top and the direction of search is clockwise (data bits rotate counter-clockwise under the head). A record containing the string Q,C,B,P,A,B,N,L,K,R,C,T,C begins at the origin of the left-most segment and continues over all four segments. The right-most segment also contains the start of the next record which consists of the string beginning B,A,C.

The first command causes all words containing the character A to be marked in the MATCH bit. Thus, after one circulation of the storage, the words are marked as shown in Figure 8b.

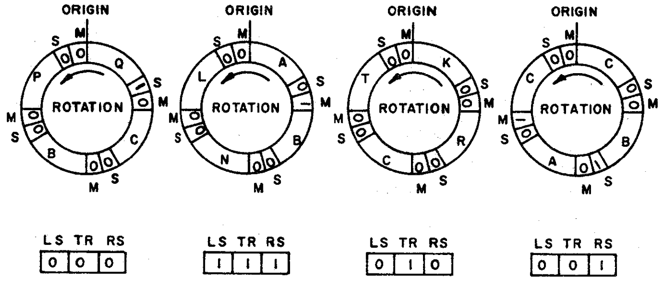


Figure 8b—Flag and status bits after search for A

In order to perform context-search operations in one storage cycle, status bits must be provided in each cell. These are used to propagate information about records which are apread over more than one cell. The status bits and their uses are:

(1) TR: The *TR*ansparent status bit is set if no word in the cell is marked with a START bit. It is used to indicate that the status indication to be transmitted to adjacent cells depends upon the status of this cell and the status input from adjacent cells.
(2) LS: The *L*eft *S*tatus bit is set if any word in a cell between the origin and the first word marked with a START bit is marked with a MATCH bit. This bit indicates a match in a record which begins to the left of the cell indicating the status.
(3) RS: The *R*ight *S*tatus bit is set if any word in the cell following the last word marked with a START bit is marked with a MATCH bit. This bit indicates a match condition which applies to words stored in the cells to the right of this cell, up to the next word marked by a START bit.

These status bits are updated at the end of each cycle of the storage. The condition of the status bits after each operation is performed is shown in Figure 8.

The second search command causes all previous MATCH bits to be erased, after using them in marking those words which contain a B and follow a previously marked word in the same record. If the previously

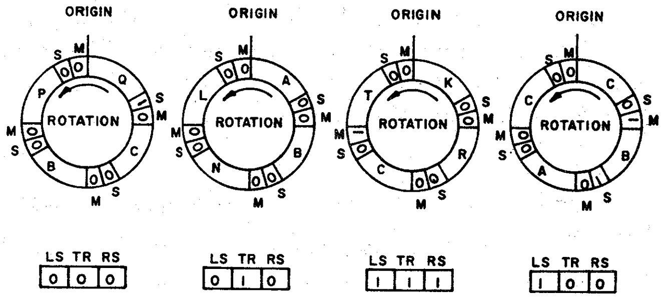


Figure 8c—Flag and status bits after search for B

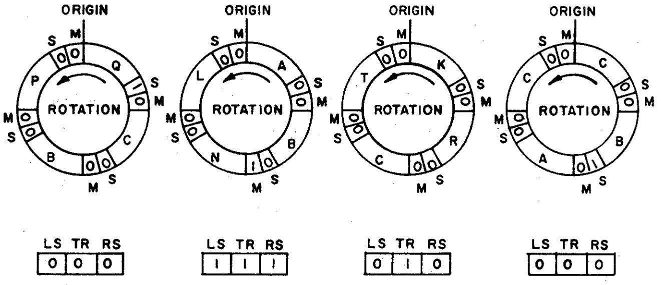


Figure 8d. Flag and status bits after search for C

marked bit and the word containing the B are in the same cell, the marking condition is completely determined by the logic in the cell. However, in most cases it is necessary to sense the status bits of previous cells in order to determine whether the ordered search condition is satisfied. Notice that the status bit conditions can be propagated down a chain of cells in the same manner as a high-speed carry is propgated in a conventional parallel arithmetic unit.

Figure 8c shows the flag-bit configurations for each word in storage and the status bits for each cell after the completion of the search for B. Figure 8d shows the configurations after the C search. After three cycles of the storage, all records in storage have been searched and those containing the ordered set of characters A, B, C have been marked. In general, a search requires one storage cycle per character in the search specification and is independent of the total storage size.

BASIC OPERATIONS

In this section, the operations for performing context searches are described in a more formal manner than in the example above. The instructions are a subset of the complete set which is described in a report.[10] The use of these instructions will be illustrated in the section following this one.

Each instruction includes a basic operation type and, in most cases, a function code which further specifies what the instruction is to do. Figure 9 shows the instruction format and its variations. Instructions which perform search and mark operations use the function code to specify the type of comparison to be used. Instructions which initiate input or output operations allow two specifications in the function field. The first designates the channel to be used in the data transfer. The second tells whether the start of each record should be marked, in preparation for another search operation.

INST TYPE	COMPARAND	FUNCTION

Figure 9a—Basic instruction format

SEARCH & MARK	COMPARAND	COMPARISON TYPE

Figure 9b—Search and mark instruction format

The symbols used in describing the instructions are given below. The notation is that due to Iverson, modified for convenience in describing some of the special operations performed by the search logic.

B: The contents of the Bit-length Register is denoted $\underline{B}$. The word length $b = \perp \underline{B}$.

C: The contents of the Comparand Register is denoted $\underline{C}$. Individual bits are c_1 (least significant bit) through c_b (most significant bit).

K: The contents of the Mask Register is denoted $\underline{K}$. Individual bits are represented by the same scheme as that used for $\underline{C}$.

W: The word of cell storage currently being considered is denoted $\underline{W}$. Individual bits are represented by the same scheme as that used for $\underline{C}$.

R: R denotes the contents of a flip-flop in each cell which is used to indicate the result of the comparison. $R \leftarrow 1$ for a "match" and $R \leftarrow 0$ for "no match". The match performed is the comparison between $\underline{C}$ and $\underline{W}$ in those positions where $k_i = 1$. In the examples considered in this paper, the comparisons are arithmetic $(=, \leq, \geq)$.

M: The MATCH bit associated with each word (see Figure 7) is denoted M. M without superscript designates the MATCH bit in the word being compared, $\underline{W}$. M with a numeric superscript indicates the MATCH bit before or after the one being compared; e.g., M^{-2} represents the MATCH bit two words before the word on which the comparison is being made. Inequality signs used as superscripts indicate logic signals representing the union of

INPUT—OUTPUT	NOT USED	CHANNEL NUMBER	*

* INDICATES THE START FUNCTION BIT

Figure 9c—Input-output instruction format

TABLE I—Description of Instructions

SS C String Search
$M \leftarrow (R \wedge M^{-1}) \vee (M \wedge P)$
Set the MATCH bit in any word where the masked comparison of the word and the comparand satisfies the comparison type specified in the function field of the instruction and the word is immediately preceded by a word in the same record which was left with its MATCH bit set by the previous instruction. Also, set the MATCH bit in any word which was left with its MATCH bit set by the previous instruction and has its PERMANENT bit set. Reset all other MATCH bits.

OS C Ordered Search
$M \leftarrow (R \wedge M^{<}) \vee (M \wedge P)$
Set the MATCH bit in any word where the masked comparison of the word and the comparand satisfies the comparison types specified in the function field of the instruction and the word is preceded (not necessarily immediately) by a word in the same record which was left with its MATCH bit set by the previous instruction. Also, set the MATCH bit in any word which was left with its MATCH bit set by the previous instruction and has its PERMANENT bit set. Reset all other MATCH bits.

MS — Mark Start
$w_i \leftarrow S \wedge (M^{>} \vee M)$ where $i = \perp$(Channel No.)
$M \leftarrow S \wedge$(Start Function)
If the channel number i specified in the instruction is between 1 and b, set w_i, the ith bit of the first word in any record which contains a word with its MATCH bit set. If the start function bit in the instruction is a one, set the MATCH bit in any word which has its START bit set. Reset all other MATCH bits.

all MATCH bits in the record before ($M^{<}$) and after ($M^{>}$) the word being compared.

P: The PERMANENT bit associated with each word (see Figure 7) is denoted P. The same superscript conventions apply to P as to M.

S: The START bit associated with each word (see Figure 7) is denoted S. The same superscript conventions apply to S as to M.

The instructions which are considered in the examples in the next section are described in Table I.

SEARCH EXAMPLES

The following examples show the application of the segment-sequential storage to matching strings with templates.[11] A template consists of characters separated by parameter markers which are to be matched by parameter strings. For example, $AB$CD$EF$ is a template which matches any string formed by the concatenation of any arbitrary string, the string AB, another arbitrary string, the string CD, another arbitrary string, the string EF, and another arbitrary string.

TABLE II—Data Format for String XYABLMNCDPEFWZ

WORD	CONTENTS
1	I/O Flags (S)
2	X
3	Y
4	A
5	B
6	L
7	M
8	N
9	C
10	D
11	P
12	E
13	F
14	W
15	Z

(S) indicates the START bit for this word is set.

TABLE III—Program to Find Match for $AB$CD$EF$

NO.	INSTRUCTION TYPE	FUNCTION	COMPARAND	REMARKS
1	OS	=	A	mark all strings which begin A or $.
2	SS	=	B	mark all strings which begin AB, $, or $B.
3	OS	=	C	mark all strings which follow the above strings and begin C or $.
4	SS	=	D	mark all strings which satisfy the AB search and contain a subsequent string which satisfies the CD search.
5	OS	=	E	mark all strings which follow the above strings and begin E or $.
6	SS	=	F	mark all strings which satisfy the template.
7	MS	2,S	—	flag channel #2 for input and mark the start of each record.

The arbitrary strings need not be the same, and any or all may be the null string. The string XYABLMNCDPEFWZ is one example of a string which matches this template.

In the following examples, it is assumed that the first word in each record has had its MATCH bit set by the last instruction of the previous search. The programs shown perform the specified search, initiate the input of the selected records to the computer, and mark the first word of each record in preparation for the next search.

Find strings to fit a template

The case where a set of fixed strings is stored in the segment-sequential storage is illustrated first. The data format for a typical string is shown in Table II. The first word is used to hold I/O flags. The characters in the string are stored in sequential words following the I/O word.

The program to search all strings in storage and mark the ones that match the template $AB$CD$EF$ is shown in Table III. A template search takes one instruction for each character in the template plus an instruction to set the I/O flag in those records which contain the strings matching the template.

Find templates to fit a string

The case where a set of templates is stored in the segment-sequential storage is considered next. The data format for stored templates is shown in Table IV. The parameter marker, $, is replaced in storage by use of the PERMANENT bit in those words which contain a character which is followed by a parameter marker.

A program to find templates to match the string XYABLMNCDPEFWZ is shown in Table V. The execution of this program illustrates how the PERMANENT bit is used. The X and Y searches do not find a match with the template shown in Table IV. However, since the PERMANENT bit in the first word in the record is set, the first word will remain marked by a MATCH bit and therefore continue as a candidate for a successful search.

TABLE IV—Data Format for Template
$AB$CD$EF$

WORD	CONTENTS	
1	I/O Flags	(S),(P)
2	A	
3	B	(P)
4	C	
5	D	(P)
6	E	
7	F	(P)

(S) indicates the START bit for this word is set.
(P) indicates the PERMANENT bit for this word is set.

TABLE V—Program to Find Templates for
XYABLMNCDPEFWZ

NO.	INSTRUCTION TYPE	FUNCTION	COMPARAND	REMARKS
1	SS	=	X	mark all strings which begin X or $.
2	SS	=	Y	mark all strings which begin XY or $.
3	SS	=	A	
4	SS	=	B	
5	SS	=	L	
6	SS	=	M	
7	SS	=	N	
8	SS	=	C	
9	SS	=	D	
10	SS	=	P	
11	SS	=	E	
12	SS	=	F	
13	SS	=	W	
14	SS	=	Z	
15	MS	1,S	—	flag channel #1 for input and mark the start of each record.

The A and B searches cause the MATCH bit in the word containing B to be set. Since this word also has its PERMANENT bit set, the MATCH bit will remain set during the searches for the remaining characters in the input string (except for the last character). The search continues in this fashion, with MATCH bits associated with characters immediately followed by a parameter marker being retained. This results in multiple string searches within each record, corresponding to different ways a given string may fit a template.

The search process continues in this fashion up to the last character in the input string. There are two ways in which a template can satisfy this search: (1) the last character in the template may match the last character in the input string and the next-to-last character in the template have its MATCH bit set, or (2) the last character in the template may have both its MATCH bit and its PERMANENT bit already set. The last search instruction in the program tests for both these conditions and at the same time resets the MATCH bits in all characters which do not meet the conditions. The last instruction in the program causes the records which satisfy the search to be marked for input to the computer's core storage.

The examples above show that the segment-sequential storage reduces the finding of matching templates to a simple search. The time required to execute such a search depends only upon the number of characters in the query.

Examples of other possible applications of the segment-sequential storage are given in a report.[10] One use is retrieval of information necessary to display a portion of a map. This is a typical problem encountered in graphic displays, where a subset of the data base is to be selected on the basis of x−y location. Another example is the use of the segment-sequential storage to simulate networks of linear threshold devices.

CONCLUSIONS

This paper has presented a new architecture designed to solve some of the problems in searching large data bases. The examples given indicate its usefulness in several practical applications. Since the system is built around a relatively inexpensive storage medium, it is feasible now. In the future, LSI techniques should make its cellular organization even more attractive.

REFERENCES

1 P ARMSTRONG
Several patents

2 G J LIPOVSKI
On data structures in associative memories
Sigplan Notices Vol 6 No 2 pp 347–365 February 1971

3 G ESTRIN R H FULLER
Some applications for content-addressible memories
Proc FJCC 1963 pp 495–508

4 R G EWING P M DAVIES
An associative processor
Proc FJCC 1964 pp 147–158

5 G J LIPOVSKI
The architecture of a large associative processor
Proc SJCC 1970 pp 385–396

6 L HELLERMAN G E HOERNES
Control storage use in implementing an associative memory for a time-shared processor
IEEE Trans on Computers Vol C-17 pp 1144–1151 December 1968

7 P T RUX
A glass delay line content-addressed memory system
IEEE Trans on Computers Vol C-18 pp 512–520 June 1969

8 I FLORES
A record lookup memory subsystem for software facilitation
Computer Design April 1969 pp 94–99

9 G J LIPOVSKI
The architecture of a large distributed logic associative memory
Coordinated Science Laboratory R-424 July 1969

10 L D HEALY G J LIPOVSKI K L DOTY
A context addressed segment-sequential storage
Center for Informatics Research University of Florida TR 72-101 March 1972

11 P WEGNER
Programming languages, information structures, and machine organization
McGraw-Hill 1968

A cellular processor for task assignments in polymorphic, multiprocessor computers

by JUDITH A. ANDERSON

National Aeronautics & Space Administration
Kennedy Space Center, Florida

and

G. J. LIPOVSKI

University of Florida
Gainesville, Florida

INTRODUCTION

Polymorphic computer systems are comprised of a large number of hardware devices such as memory modules, processors, various input/output devices, etc., which can be combined or connected in a number of ways by a controller to form one or several computers to handle a variety of jobs or tasks.[1] Task assignment and resource allocation in computer networks and polymorphic computer systems are currently being handled by software. It is the intent of this paper to present a cellular processor which can be used for scheduling and controlling a polymorphic computer network, freeing some of the processor time for more important functions. (See Figure 1.)

Work has been done in the area of using associative memories and associative processors in scheduling and allocation in multiprocessor systems.[2,3] Since the scheduling process often involves a choice of hardware resources which might do the job, a system able to detect "*m* out of *n*" conditions being met would be more suited to the type of decision-making required. The system to be discussed involves a threshold-associative search; that is, all the associative searching performed detects if at least *m* corresponding bits in both the associative cell and the comparand are one.

Scheduling and controlling can be divided into three distinct phases. The first is task qualification, determining which tasks are possible with the available hardware. The second phase is task assignment, deciding which of the candidate tasks found to be qualified in the first phase will be chosen to be performed next. The third phase is the actual controlling or connection of the switch required to restructure the computer to perform the selected tasks.

This paper will be restricted to those functions performed by the cellular processor; in particular, the task qualification phase and the portions of the task assignment phase related to the cellular processor.

SCHEDULING

The method for ordering requests consists of storing the queue of requests in a one-dimensional array of cells. One request requires several contiguous cells for storage. The topmost cells store the oldest request. New requests are added to the bottom and are packed upward as in a first-in, first-out stack. An associative search is performed over all the words stored to determine which requests qualify for assignment. The topmost request which qualifies will be chosen for assignment. Using a slightly more complex cell structure, a priority level may be associated with each request, resulting in a priority based, rather than chronological, method for task assignment, providing for greater flexibility. The priority-based system will not be discussed here, but further detail relative to it may be found in a previous report.[4]

METHOD OF OPERATION

The basic system consists of a minicomputer and a cellular processor for task ordering. (See Figure 1.) Requests generally take the form of which processors are required, how much memory is required, and which

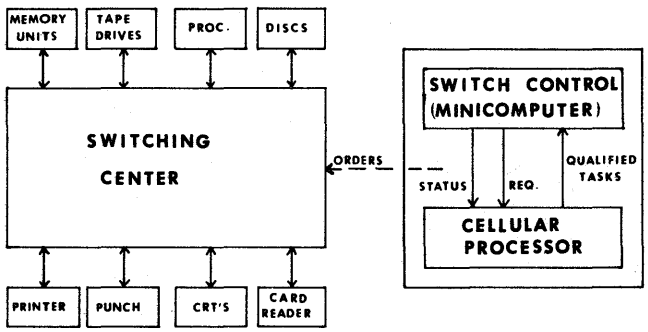


Figure 1—Polymorphic computer network controlled by cellular processor and minicomputer

peripheral devices and how many of each type are required to perform a particular task. These requests are made to the minicomputer via a simple, low-volume communication link, such as a shift register, data bus, or belt. The minicomputer then formats the requests into a *request set* which is explained below.

The request set is given an identification word and is input to the bottom of the task queue stored in the cellular processor. This unit stores all the request sets and determines which requests can be qualified for assignment based on current hardware availability. The topmost request set in the cellular processor which qualifies is chosen for assignment.

It is nécessary for the processor to know which devices in the polymorphic computer system are not currently in use, and therefore are available for assignment. To provide this information, each physical device in the system has a bit associated with it in an *Availability Status Register*. If a unit, such as a tape drive, is free, its corresponding bit in the status register will be a one. When the unit is in use, its corresponding bit will be reset to a zero.

The *requests* are of the form indicating which type of hardware devices are required, how many are required and which, if any, particular physical units are required. These requests can all be expressed as a Boolean AND of threshold functions. Each request word will correspond to one threshold function, including the threshold value. The devices chosen from to meet that threshold value will be indicated with a one in its bit position. Let S be the status register and (Q) (T) be the request word where Q is the binary vector representing a request and T the binary number giving the threshold value T. The output C of the threshold function may be expressed as

$$C \leftarrow T \leq \sum_{1}^{n} Q[I] \wedge S[I].$$

TABLE I—Status Register Assignment

BIT	DEVICE
1,2	Processors 1 and 2
3-6	Memory Units 1-4
7-12	Tapes Drives 1-6
13	Line Printer
14	Disc
15	Card Reader
16	Card Punch
17,18	CRT 1 and 2

A request set then consists of an identification word and a word for each threshold function necessary to express the entire request.

Consider, for example, a system composed of the components or peripheral devices and the status register bit assignments shown in Table I. The status register in this example would be 18 bits long.

ID WORD

BITS: 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1	0	1	1	0	0	1
								ID										THRESHOLD			

REQUEST WORDS

PROC	MEMORY	TAPE DRIVES	LP	D	CR	CP	CRT	THRESHOLD
1 0	0 0 0 0	1 0 0 0 0 0	1	0	0	0	1 0	0 1 0 0
0 0	0 0 0 0	0 1 1 1 1 1	0	0	0	0	0 0	0 0 1 0
0 0	1 1 1 1	0 0 0 0 0 0	0	0	0	0	0 0	0 0 1 0

Figure 2—Request set example

A request would be of the sort that the required devices for Task Number 429 are Processor 1, CRT 1, Tape Drive 1, and any two other tape drives, the Line Printer, and any two memory units. This request set would consist of four words, the ID word and three request words, shown in Figure 2.

The threshold value of the ID word is set exactly equal to the number of "1's" in the ID field. This is for hardware considerations in order to do an associative search on the ID words. All the units which are absolutely necessary (mandatory devices) can be compactly represented by a single threshold request that implements the AND function. The first request word represents all such mandatory devices, whereas the second and third request words represent "any two other tape drives" and "any two memory units," respectively.

This request set, along with any other requests which were made would be input to the queue. When all three of the request words above could be satisfied with some or all of the available hardware, an interrupt to the minicomputer is generated. The minicomputer can then read out the ID word of the topmost request set that can be satisfied and is therefore qualified. If this request set is the highest in the queue, it will be assigned. Whichever request set is read out will be removed from the cellular processor and the requested resources allocated for that task by the minicomputer.

HARDWARE DESCRIPTION

The hardware realization for this cellular processor consists of a bilateral sequential iterative network.[5] That is, it is a one-dimensional array of cells, all cells having the same structure and containing sequential logic. Each cell receives external inputs as well as inputs which are outputs from its adjacent cells as shown in Figure 3.

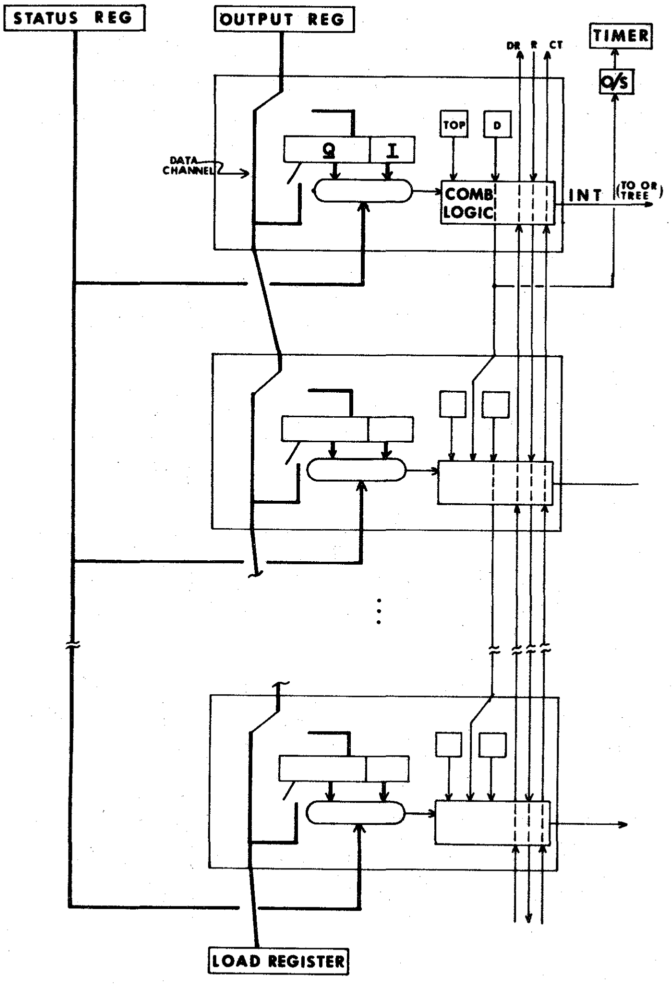


Figure 3—Cellular processor

Each cell stores either a request word or an ID word, or it is empty. All cells receive hardware status information which is broadcast into them continuously for comparison with their stored requests. When one or more request set has qualified for assignment, an interrupt is generated to the minicomputer. A hardware priority circuit chooses the topmost qualified request to be assigned. The cellular processor outputs the ID word for this request via a data channel which is set up through all the cells above the cell containing the qualified request in the queue. When a request is chosen for assignment, its ID word is broadcast to the cellular processor for removal from the queue. A timer is associated with the uppermost cell in the array and is used to indicate if requests are stagnating in the queue so that action may be taken by the minicomputer.

Requests are always loaded into the bottom of the queue. Removal is either from the top, when the timer mentioned above exceeds some maximum value, or by deletion after the request has been assigned. If a task request is cancelled, it may be removed from the queue by treating it as if it were assigned. When requests are removed from the middle of the queue by assignment, the other requests move upward to pack into the emptied cells.

Each cell is basically made up of an n-bit register, a threshold comparator, two cell status flip flops, a data channel, and combinational logic as shown in Figure 4. The n-bit register is divided into two fields. The first k bits, Q, store the binary vector representation of the request or ID word. The last $n-k$ bits represent the threshold value, T, for the threshold comparator. The threshold comparator, which will be discussed in more detail later, outputs a one if and only if at least T positions in both Q and the status input, S, are one's. That is,

$$C \leftarrow T \leqq \sum Q[I] \wedge S[I] \quad \text{or,}$$

$$C \leftarrow 0 \leqq \left(\sum Q[I] \wedge S[I]\right) - \left(\sum T[I] \times 2^{I}\right).$$

The two cell status flip flops, TOP_{ff} and D_{ff} indicate whether a cell contains an ID word or not and whether a cell contains data or is empty, respectively.

The data channel through each of the cells is used to output information and for packing data to economize on the number of pins per cell. The data flow in the data channel is always upward, toward the top of the queue. The data channel within a cell may be operated in two ways. It may allow data coming into the cell on

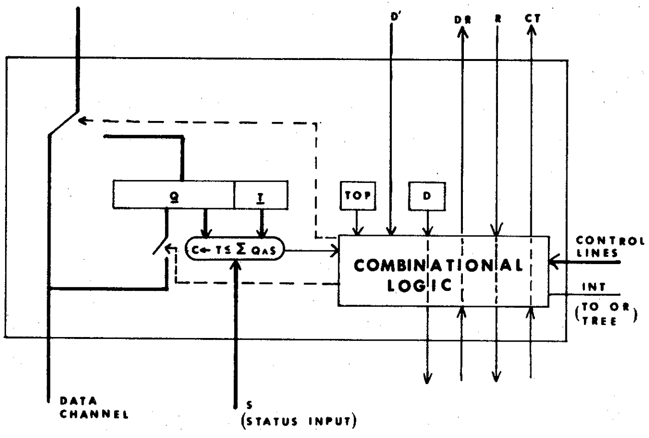


Figure 4—Basic iterative cell

the channel to pass through into the data channel of the next cell and will be referred to as the bypass mode of operation. Also, by means of an electronic switch, it may place the contents of its register into the data channel. This will be referred to as the transfer mode. Through the use of the load enable of the register (the clock input of the register flip flop), it is also possible to load the register with the information which is in the data channel. Operation of the data channel is controlled by the cell status, the control signals from the minicomputer, and a compare rail, CT.

When a request is loaded into the cellular processor, it enters via the data channel and is loaded adjacent to the lowest cell containing data. This is determined by the D_{ff} output from the cells. Once a cell has data loaded, its threshold comparator continuously compares the register contents, Q, against the status, S. When a threshold compare has been achieved, that is,

$$T \leqq \sum S[I] \wedge Q[I]$$

a one is ANDed into the CT rail, which is propagating upward, toward the top of the queue. When all the request words in a set compare, the CT rail entering the TOP cell of the request set is a logic one. This causes an interrupt to be generated, indicating to the minicomputer that there is a qualified set. The interrupt, INT, is placed into an OR tree external to the cell network to speed the interrupt signal to the minicomputer to increase response time of the system. Upon receipt of the interrupt, the minicomputer can interrogate the processor to determine which request set caused the INT to be generated. The ID word of the topmost qualified set is broadcast via the data channel, and stored in the output register. The minicomputer can then remove the request set from the queue by placing the ID of that set on the status lines and commanding a set removal via the control lines. While a removal is being commanded, the set whose ID matches with the ID on the status lines resets its data flip flop, D_{ff}, and passes a one along the R (reset) rail. This rail propagates in a downward direction and causes all cells to reset their D_{ff} until a TOP cell is encountered. This removes the request set from the queue. There now is a group of empty cells in the middle of the stack of cells. When a cell containing data detects an empty cell above it, it places its data into the data channel and generates a pulse on the DR (data ready) rail. This pulse travels upward and enables the loading of data into the uppermost cell in the group of empty cells, that is, the first empty cell below a non-empty cell which it encounters. This is determined by D', the value of the D_{ff} of the next higher cell. Each cell moves its data upward until all the empty cells are at the bottom of the queue.

The comparison operation is not stopped by the data being in the process of packing. The compare rail, CT, is passed through empty cells unless the DR rail is high, indicating data is actually in transit. An example of the switching of the data channel during the loading and shifting, or packing, process is shown in Figure 5.

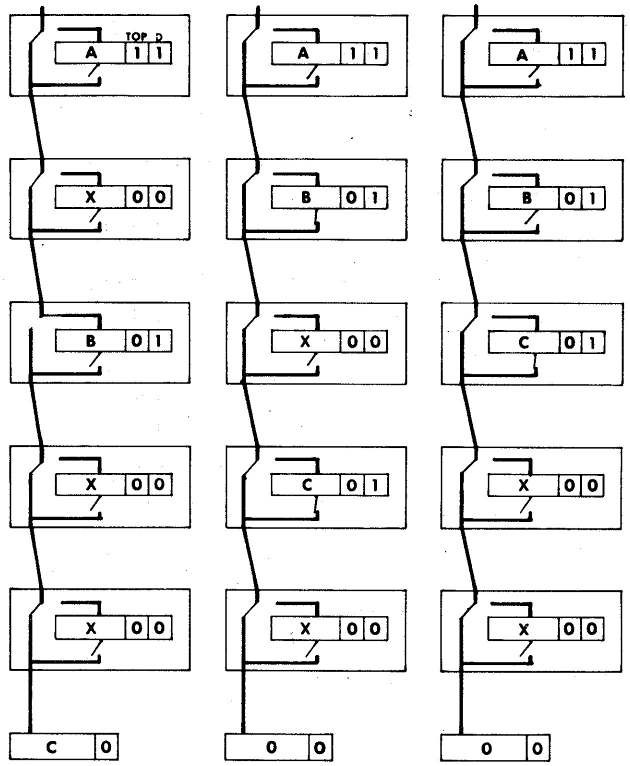
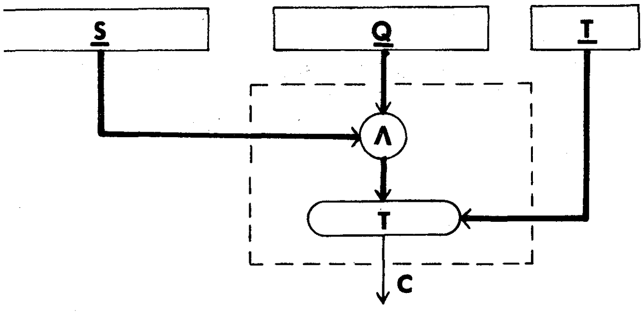


Figure 5—Example of shifting and loading

Further details of the cell operation are given in an earlier report.[4] A method for implementing priority handling was also discussed.

THRESHOLD COMPARATOR

Current literature on threshold logic discusses integrated circuit realizations of threshold gates with up to 25 inputs and with variable threshold values.[6,7] The threshold comparator mentioned earlier consists of a threshold gate with variable threshold which is selected by the contents of the threshold register. The inputs to the threshold gate are the contents of the status register, S, ANDed bit by bit with the contents of the cell request register, Q, as shown in Figure 6. All inputs are weighted one.



$$C \leftarrow T \leq \sum S[I] \wedge Q[I]$$

Figure 6—Threshold comparator

If the number of inputs to the threshold gate is restricted to the 25 inputs indicated above, the hardware realization discussed here must be modified to overcome this restriction. In particular, the various types of resources can be divided into disjoint sets of similar or identical devices such as memory units, processors, I/O devices, etc. A request would not be made, for instance, which would require either a tape drive or a processor. Each set would then have a threshold value associated with it and the compare outputs from all the threshold gates would be ANDed to yield the cell compare output, as illustrated in Figure 7. For simplicity, we will assume an ideal threshold element exists with an unlimited number of gate inputs in our further discussion, which can be replaced as indicated above.

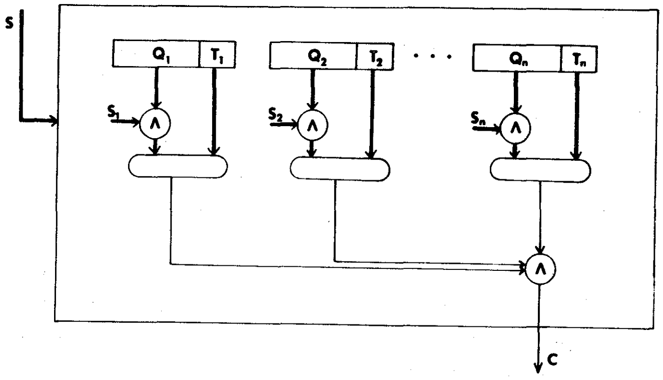


Figure 7—Modular threshold comparator

For large computer networks, the number of devices will be large. Since the processor discussed here requires more than $3n$ interconnections (pins) for each cell, where n is the number of devices, a method of dividing the cell into smaller modules which are feasible with current technologies in LSI must be considered.

First, the cell must be split into modules of lower bit sizes. This may be done as discussed previously by dividing the hardware devices into disjoint sets of similar or identical devices. Each module or sub-cell will then have a threshold associated with it and a threshold comparator. One control sub-cell is also necessary which will contain all the logic required for storing the cell status, generating and propagating the rail signals, and control the data channels in the other sub-cells in its cell group. This is illustrated in Figure 8.

This modularity of cell design also allows the cellular processor to be expandable. If the system requirements demand a larger (more bit positions) cell, rather than having to replace the entire cellular processor, an additional storage module may be added for each cell. This also reduces the fabrication cost since only two cellular modules would have to be designed regardless of the number of devices in a system.

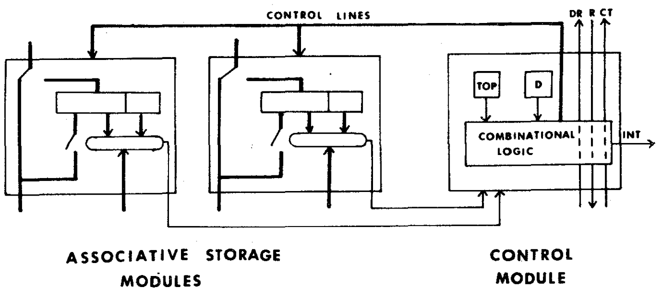


Figure 8—Modular cell structure

CONCLUSION

The threshold associative cellular processor incorporates a very simple comparison rule, masked threshold comparison. This rule was shown to be ideally suited to task qualification in a polymorphic computer, or an integrated computer network like a polymorphic computer, and was shown to be easily implemented in current LSI technology.

The processor developed using this type of cell would considerably enhance the cost effectiveness of polymorphic computers and integrated computer networks by performing task requests and would reduce the software support otherwise required to poll the status of devices in the polymorphic computer or an integrated computer network. The scheme shown here will have application to other task qualification problems as well, such as a program sequencing scheme to order programs or tasks based on a requirement for previous tasks to have been performed.[4] This modular cellular processor provides a system which can handle a wide range of scheduling problems while retaining a flexibility for expansion and at the same time increasing speed by performing the parallel search rather than polling.

REFERENCES

1 H W GSCHWIND
Design of digital computers
Chapter 9 Springer Verlag 1967

2 D C GUNDERSON W L HEIMERDINGER J P FRANCIS
Associative techniques for control functions in a multiprocessor, final report
Contract AF 30(602)-3971 Honeywell Systems and Research Division 1966

3 D C GUNDERSON W L HEIMERDINGER J P FRANCIS
A multiprocessor with associative control
Prospects for Simulation and Simulator of Dynamic Systems Spartan Books New York 1967

4 J A ANDERSON
A cellular processor for task assignments in a polymorphic computer network
MS Thesis University of Florida 1971

5 F C HENNIE
Finite state models for logical machines
John Wiley & Sons New York 1968

6 J H BEINART et al
Threshold logic for LSI
NAECON Proceedings May 1969 pp 453-459

7 R O WINDER
Threshold logic will cut costs especially with boost from LSI
Electronics May 27 1968 pp 94-103

A register transfer module FFT processor for speech analysis

by DAVID CASASENT and WARREN STERLING

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

On-line speech analysis systems are the subject of much intensive research. Spectral analysis of the speech pattern is an integral part of all such systems. To facilitate this spectral analysis and the associated preprocessing required, a special purpose fast Fourier transform (FFT) processor to be described is being designed and constructed. One unique feature of this processor which facilitates both its design and implementation while providing an easily alterable machine is its construction from standard logic modules which will be referred to throughout as register transfer modules or RTM's.[1] This design approach results in a machine whose operation is easily understood due to this modular construction.

Two of the prime advantages of such a processor are:

(1) The very low design, implementation, and debugging lead times which result from the RTM design at the higher register transfer logic level rather than at the conventional gate level.
(2) The RTM processor can be easily altered due to the pin-for-pin compatability of all logic cards. Different hardwired versions of a given algorithm can be easily implemented by appropriate back plane rewiring.

Because of the stringent time constraints imposed by such a design effort, this processor can also serve as a feasibility model for the use of RTM's in other complex real-time systems. This is one area in which little work has been done.

When in operation, the processor will accept input data in the form of an analog speech signal and output the resultant spectral data to a PDP-11 computer for analysis.

FOURIER TRANSFORM APPLICATIONS TO SPEECH PROCESSING[2]

Let us briefly review Fourier transform techniques as used in speech processing.

In the discrete time domain, a segment of speech $s(\xi T+nT)$ can be represented by

$$s(\xi T+nT)=p(\xi T+nT)*h(nT) \qquad (1)$$

where $*$ denotes discrete convolution and ξT is the starting sample of a given segment of the speech waveform. $p(\xi T+nT)$ is a quasiperiodic impulse train representing the pitch period and $h(nT)$ represents the triple discrete convolution of the vocal-tract impulse response $v(nT)$, with the glottal pulse $g(nT)$ and radiation load impulse response $r(nT)$,

$$h(nT)=v(nT)*r(nT)*g(nT) \qquad (2)$$

The vocal tract impulse response is characterized by parameters called formant frequencies. These parameters vary with corresponding changes in the vocal tract as different sounds are produced; however, for short time spectrum analysis of speech waveforms, the formant frequencies can be considered constant.

Given the above speech model, speech analysis involves estimation of the pitch period and estimation of formant frequencies. These parameters are estimated using the cepstrum of a segment of a sampled speech waveform. For our purposes, the cepstrum is defined as the inverse discrete Fourier transform (IDFT) of the log magnitude spectrum of the speech waveform segment. The details of cepstral analysis are shown in Figure 1. The input speech segment to Figure 1 $s(\xi T+nT)$, typically about 20 msec in duration, is weighted by a symmetric window function $w(nT)$

$$\begin{aligned} x(nT)&=s(\xi T+nT)w(nT)\\ &=[p(\xi T+nT)*h(nT)]\cdot w(nT) \quad 0\leq n<N \qquad (3)\end{aligned}$$

where N is the number of samples of the speech wave-

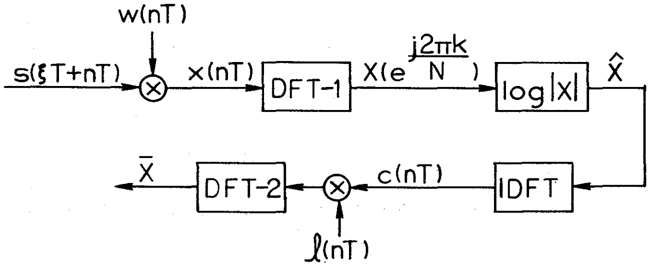


Figure 1—Cepstral analysis for formant frequency determination
$\hat{X}$ denotes log magnitude spectrum
$\bar{X}$ denotes cepstrally smoothed log magnitude spectrum

form. The window function minimizes the effect of a nonintegral number of pitch periods in each speech segment by de-emphasizing the samples at both ends of the segment. A typical window used for this purpose is the Hamming window defined by

$$w(nT)=\begin{cases}0.54-0.46\cos(2\pi nT/NT) & 0\le nT\le NT\\ 0 & \text{elsewhere}\end{cases} \tag{4}$$

With $w(nT)$ slowly varying with respect to $s(nT)$,

$$x(nT)\approx h(nT)*p_w(nT) \tag{5}$$

where

$$p_w(nT)=p(\xi T+nT)w(nT) \tag{6}$$

After the first Fourier Transform (DFT-1) the speech spectrum becomes

$$|X(e^{j(2\pi k/N)})|=|H(e^{j(2\pi k/N)})|\cdot|P_w(e^{j(2\pi k/N)})| \tag{7}$$

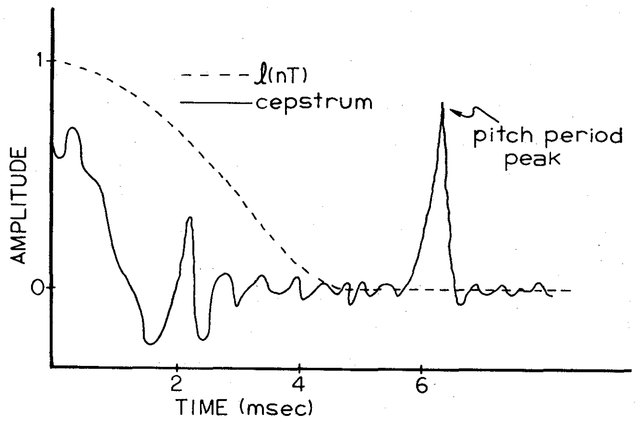


Figure 2—(a) The cepstrum

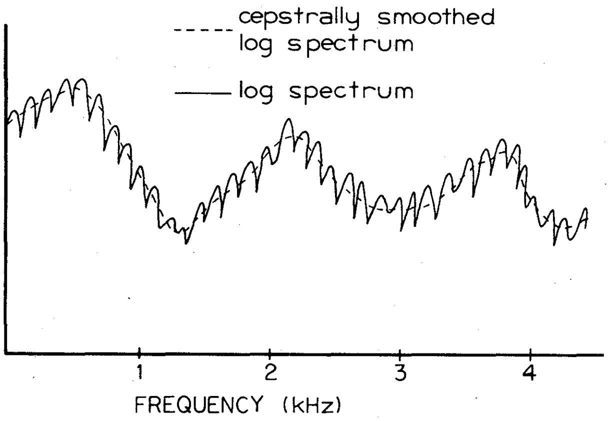
The log magnitude operation ($\log|X|$) then yields

$$\log|X(e^{j(2\pi k/N)})|$$
$$=\log|H(e^{j(2\pi k/N)})|+\log|P_w(e^{j(2\pi k/N)})| \tag{8}$$

The inverse transform (IDFT) of this log magnitude spectrum is the cepstrum $c(nT)$. The pitch period corresponding to a distinct peak in the cepstrum is removed by multiplying $c(nT)$ by a function $l(nT)$ of the form

$$l(nT)=\begin{cases}1 & |nT|<\tau_1\\ \tfrac{1}{2}\{1+\cos[\pi(nT-\tau_1)/\Delta\tau]\} & \tau_1\le|nT|<\tau_1+\Delta\tau\\ 0 & |nT|\ge\tau_1+\Delta\tau\end{cases} \tag{9}$$

where $\tau_1+\Delta\tau$ is chosen less than the minimum pitch period expected. The final Fourier transform (DFT-2) then yields the desired spectral envelope.



(b) The log magnitude spectrum and spectral envelope

Figure 2a shows the cepstrum, the pitch period corresponding to the distinct peak at the right. The dotted line in Figure 2a represents the $l(nT)$ function above. Figure 2b shows the original log magnitude spectrum (solid line) and the resultant "smoothed" spectrum or spectral envelope (dotted line) from which the formant frequencies can be estimated.

APPROXIMATE BINARY LOGARITHMS[3]

The binary logarithm can easily be obtained by the following algorithm. A binary number N, can be

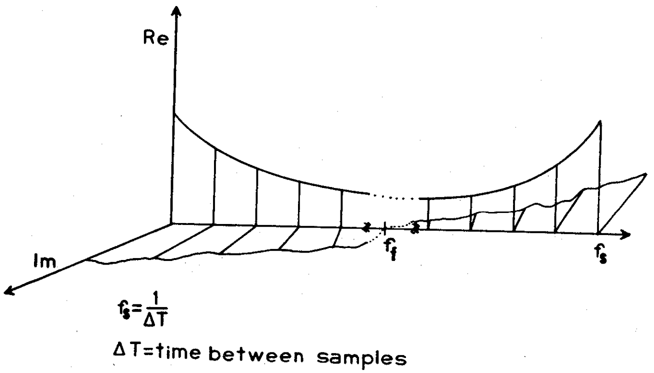


Figure 3—Symmetry of the complex discrete fourier transform for real-valued input

written as

$$N=\sum_{i=j}^{m} Z_i 2^i \tag{10}$$

where m and j represent the binary powers of the most and least significant bits respectively. If the power of the most significant nonzero bit is denoted by k, N can be rewritten as

TABLE I—Function $F(x)$ Used to Calculate Binary Logarithm of x

Range	$F(x)$
$0 \leq x < .25$	$x+\frac{37x}{128}+1/128$
$.25 \leq x \leq .50$	$x+3x/64+1/16$
$.50 \leq x \leq .75$	$x+\frac{7(1-x)}{64}+1/32$
$.75 \leq x \leq 1$	$x+29(1-x)/128$

$$N=2^k+\sum_{i=j}^{k-1} Z_i 2^i \qquad m\geq k\geq j \tag{11}$$

$$N=2^k\left[1+\sum_{i=j}^{k-1} Z_i 2^{i-k}\right]=2^k(1+x) \tag{12}$$

where $0\leq x<1$ since $k\geq j$. $\text{Log}_2\, N$ can then be approximated by $\text{Log}_2\, N\approx L(N)=k+F(x)$ where $F(x)$ is chosen from Table I.

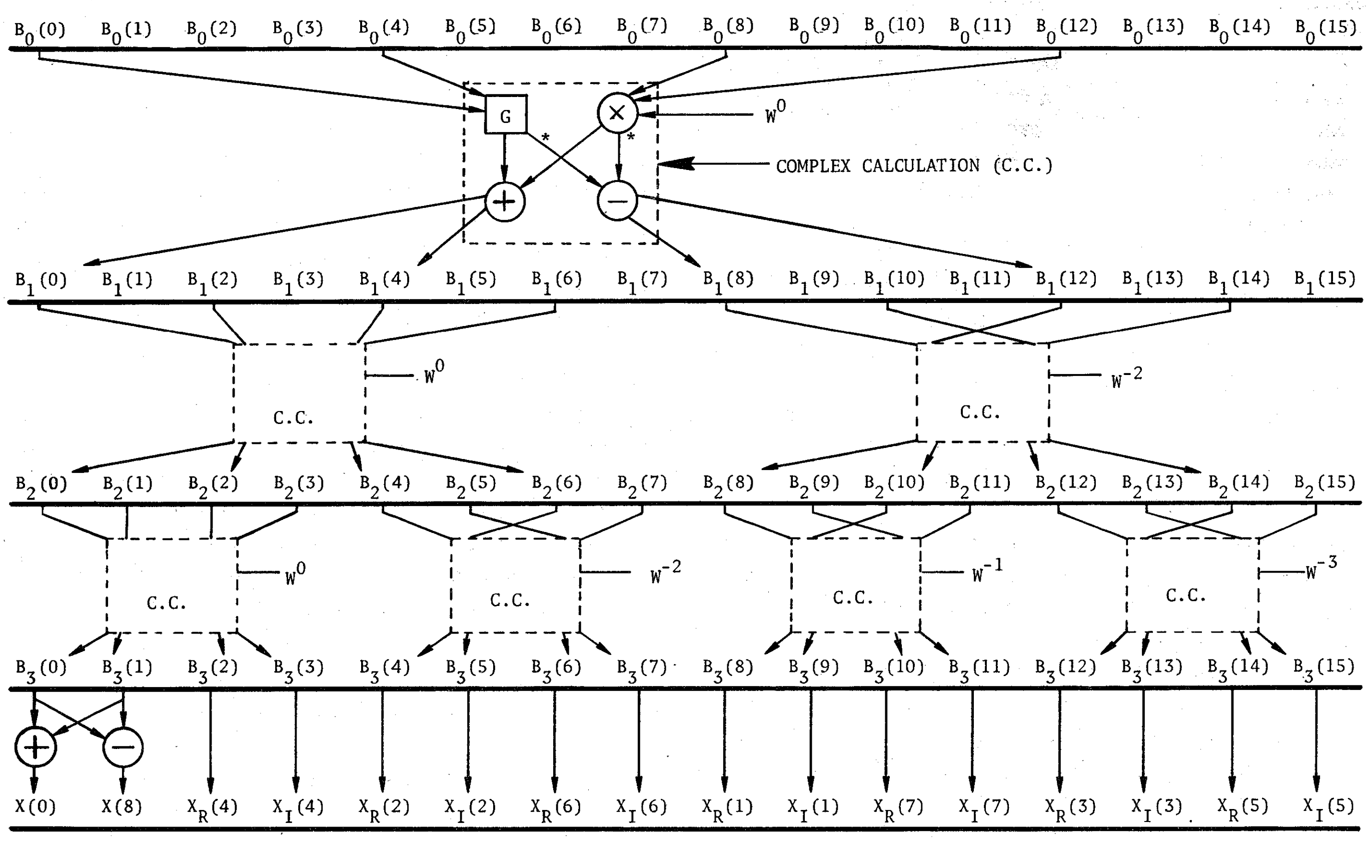
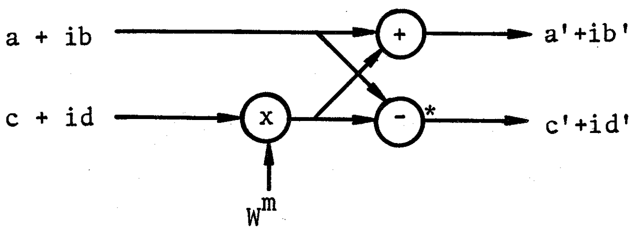

$B_0(0)$ $B_0(1)$ $B_0(2)$ $B_0(3)$ $B_0(4)$ $B_0(5)$ $B_0(6)$ $B_0(7)$ $B_0(8)$ $B_0(9)$ $B_0(10)$ $B_0(11)$ $B_0(12)$ $B_0(13)$ $B_0(14)$ $B_0(15)$
G
×
*
*
W^0
+
−
COMPLEX CALCULATION (C.C.)
$B_1(0)$ $B_1(1)$ $B_1(2)$ $B_1(3)$ $B_1(4)$ $B_1(5)$ $B_1(6)$ $B_1(7)$ $B_1(8)$ $B_1(9)$ $B_1(10)$ $B_1(11)$ $B_1(12)$ $B_1(13)$ $B_1(14)$ $B_1(15)$
W^0
C.C.
C.C.
W^{-2}
$B_2(0)$ $B_2(1)$ $B_2(2)$ $B_2(3)$ $B_2(4)$ $B_2(5)$ $B_2(6)$ $B_2(7)$ $B_2(8)$ $B_2(9)$ $B_2(10)$ $B_2(11)$ $B_2(12)$ $B_2(13)$ $B_2(14)$ $B_2(15)$
W^0
C.C.
C.C.
W^{-2}
C.C.
W^{-1}
C.C.
W^{-3}
$B_3(0)$ $B_3(1)$ $B_3(2)$ $B_3(3)$ $B_3(4)$ $B_3(5)$ $B_3(6)$ $B_3(7)$ $B_3(8)$ $B_3(9)$ $B_3(10)$ $B_3(11)$ $B_3(12)$ $B_3(13)$ $B_3(14)$ $B_3(15)$
+
−
$X(0)$ $X(8)$ $X_R(4)$ $X_I(4)$ $X_R(2)$ $X_I(2)$ $X_R(6)$ $X_I(6)$ $X_R(1)$ $X_I(1)$ $X_R(7)$ $X_I(7)$ $X_R(3)$ $X_I(3)$ $X_R(5)$ $X_I(5)$

Figure 4—The real-valued input FFT algorithm for $N = 16$ * denotes complex conjugate



$$W^m = \cos \frac{2\pi m}{N} + i \sin \frac{2\pi m}{N}$$

$$a' = a + (c \cos \frac{2\pi m}{N} + d \sin \frac{2\pi m}{N})$$

$$b' = b + (d \cos \frac{2\pi m}{N} - c \sin \frac{2\pi m}{N})$$

$$c' = a - (c \cos \frac{2\pi m}{N} + d \sin \frac{2\pi m}{N})$$

$$d' = - [b - (d \cos \frac{2\pi m}{N} - c \sin \frac{2\pi m}{N})]$$

Figure 5—The complex calculation
* denotes complex conjugate. N = number of samples

Computer calculations using this algorithm yielded a maximum error computed at critical values and extrema which ranges between −0.00782 and 0.0094. The coefficients of $f(x)$ were chosen for easy of binary implementation.

FFT ALGORITHM FOR REAL-VALUED INPUT

Various FFT algorithms exist. One particularly adaptable to RTM implementation will be briefly reviewed. The complex discrete Fourier transform of a sampled time series $x(k)\,(k=0, \ldots, N-1)$ can be written as

$$X(j) = \frac{1}{N} \sum_{k=0}^{n-1} x(k) e^{-i2\pi jk/N} \qquad (13)$$

It has been shown[4] that when the $x(k)$ series is real, Re $[X(j)]$ is symmetric about the folding frequency F_f; and Im $[X(j)]$ is antisymmetric about F_f. Figure 3 shows this pictorially.

An algorithm[5] which eliminates calculations that will lead to redundant results in the real-valued input case has previously been discussed. Figure 4 graphically illustrates this algorithm for $N=16$. The algorithm can be represented by the expression

$$X(j) = \sum_{k=0}^{n-1} B_0(k) W^{-jk} \qquad (14)$$

where $W=e^{2\pi i/N}$; $B_o(k)$ is real; $j=0, 1, \ldots, N/2$; and $N=2^m$ where m is an integer.

The "complex calculation" shown in Figure 4 is a slight modification of the butterfly multiply[6] normally used in FFT algorithms. Details of the calculation are shown in Figure 5, from which the signal flow is apparent. Each complex calculation box, as shown, moves to the right to operate on all operands within its group. On the first level, this box performs eight computations, on the second level each box performs 4 calculations, etc.

Since the multiplications are ordered as above, addressing for this multiplier is fairly straightforward. For ease in accessing the complex multiplier W^m, its complex values should be stored in the order in which they occur. An algorithm for determining the sequence of the exponent m has been documented, and a set of recursive equations which specify the addresses of the four operands for every complex calculation can be formulated.[5] The address sequencing is easily implemented in a hardware unit for automatic generation of the required addresses in the proper sequence.

It is apparent from Figure 4 that all complex calculations involving one complex multiplier W^m can be completed before the next complex multiplier is used.[7] For example, all calculations involving W^0 can be completed on all 3 levels, then all calculations involving W^2, etc. In the conventional method all calculations on one level are completed before dropping to the next level. If the complex multipliers are stored in their accessed order, there is no need to explicitly store the sequence of exponents. Furthermore, each complex exponent in this addressing scheme need be accessed only once.

As in the conventional FFT implementation, the resultant Fourier coefficients must be re-ordered. With the accessing order of the complex multipliers specified by a linear array A, the exponent m for the ith W is given by $m=A(i)$. An inverse table look-up enables the scrambled Fourier coefficients to be accessed from memory in the order of ascending frequency. To implement this inverse table look-up, the location N of the ith harmonic is found from the value m in the array A and by using its position in the array as the value of N.

TABLE II—Formulas for Calculating the Number of Operations in FFT Algorithms

	Real Multiplications	Real Additions
complex inputs	$(m - 3.5)N + 6$	$(1.5m - 2.5)N + 4$
real inputs	$(2m - 7)N + 12$	$(3m - 3)N + 4$

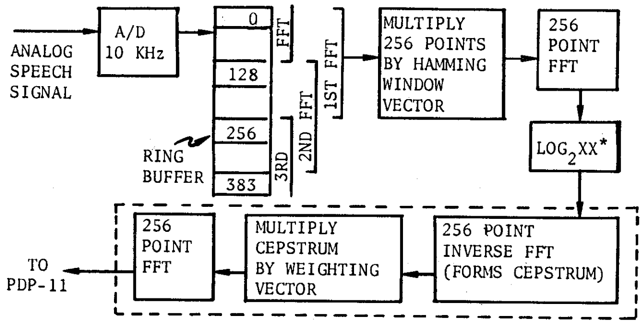


Figure 6—FFT processor data flow. Boxed area denotes future extension of the processor

In implementation, the sequence of locations is, for convenience, stored separately.

Table II below compares the number of operations, and consequently, the speed, of the conventional Cooley-Tukey radix-2 FFT algorithm for complex inputs, and the FFT algorithm for real inputs.[5] In the formulas $N=2^m$, where N is the number of samples. These formulas assume special cases such as exp ($i0$) are calculated as simply as possible. About ½ the number of operations are required for real inputs as for complex inputs, owing to the elimination of redundant calculations. As explained previously, the algorithm can be streamlined further by sequencing through the complex multipliers rather than across each level. A software version of these techniques has been implemented[7] and has achieved a real-time processing speed of 10,300 samples/sec. This is the equivalent of one 256 point FFT every 25 msec. The minimum speech processing speed required for this system is one 256 point FFT every 10 msec. It is evident that speeding up the algorithm requires hardwiring the complex calculation and address generation.

PROCESSOR DATA FLOW

Figure 6 shows the logical flow of data through the processor. The "Future Extension" section will not be implemented initially. Instead the log magnitude of the spectrum will be transferred to a PDP-11. At this point the spectral envelope can be extracted by digital recursive filtering techniques rather than by cepstral smoothing. This approach adequately demonstrates the feasibility of a real-time RTM processor.

The analog speech signal is sampled at 10 kHz and stored in a buffer. When 256 8-bit words have been accumulated, they are weighted by a Hamming window. A 256 point FFT is then performed on these weighted samples. This results in only 129 complex values since the FFT algorithm for real-valued inputs generates harmonics only through the folding frequency. The binary logarithm of the magnitude of each of these 129 complex values is then calculated and the result transferred to a PDP-11.

TABLE III—Description of RTM Modules

Module	Function
K.bus	controls asynchronous timing of sequential operations
T.a/d	analog to digital converter
DM.bool	boolean flags
DM.const	4 word read only memory
DM.gpa	general purpose arithmetic unit
DM.ii	general purpose input interface
DM.index	FFT address generator
DM.mult	multiply unit
DM.oi	general purpose output interface
DM.pdp-11	PDP-11 interface
DM.tr	temporary storage register
M.array	read/write memory; ~2 μsec access time
M.sp	read/write scratch pad memory; ~500 $n$sec access time

During processing, the buffer must continually store the input samples. After the third group of 128 samples has been stored, samples 128 thru 383 are weighted by the window and processed. Although a 256 point FFT is performed, the window is shifted by only 128 words each time thus including each sample in 2 FFT calculations, each time with a different weighting factor.

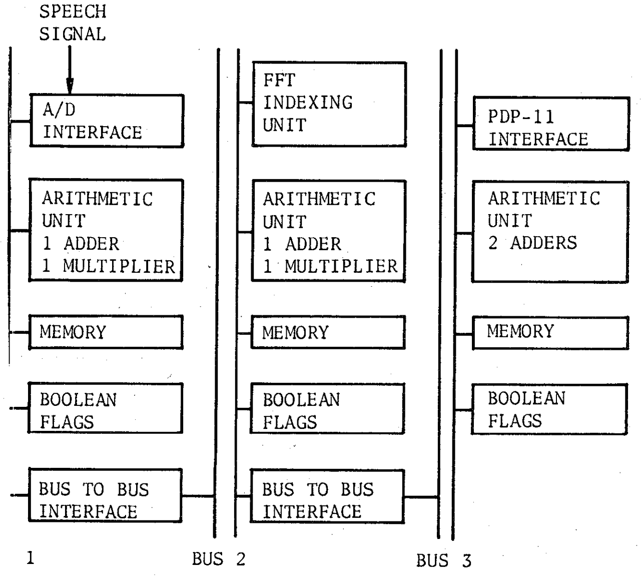


Figure 7—Block diagram of FFT processor
Bus 1 samples and buffers speech signal. Bus 2 performs FFT. Bus 3 calculates binary logarithm and interfaces to a PDP-11

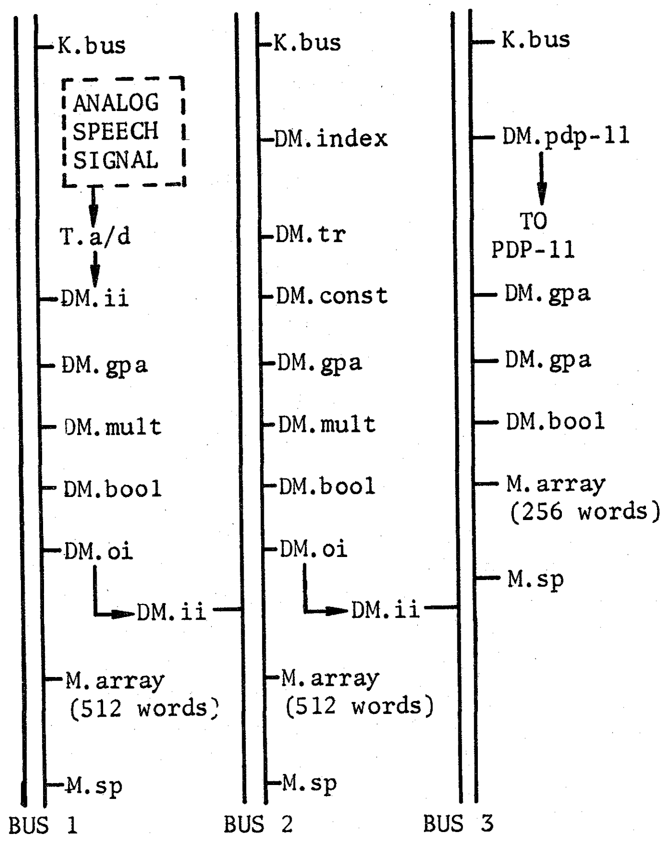


Figure 8—RTM structure of FFT processor. The modules are described in Table III

The first FFT thus operates on samples 0-255, the second FFT on samples 128-383, the third on 256-511, etc. In the actual machine a 384 word ring buffer memory is used to achieve the sequencing of the blocks of 128 samples.

The time constraints on the system are easily tabulated. In the 12.8 msec used to sample 128 words the following three operations must be performed:

(1) The Hamming window must be applied,
(2) The 256 point FFT performed, and
(3) The log magnitude of each harmonic calculated.

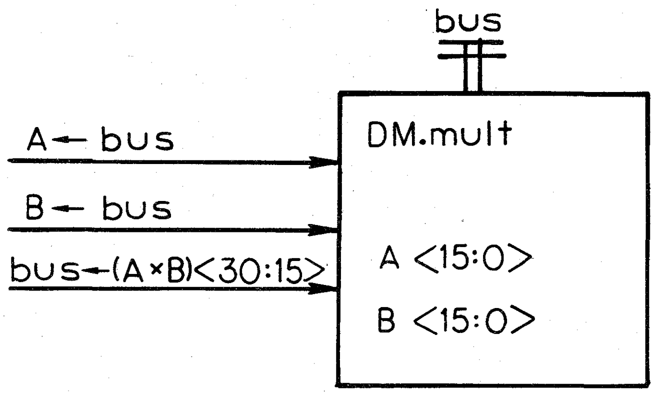
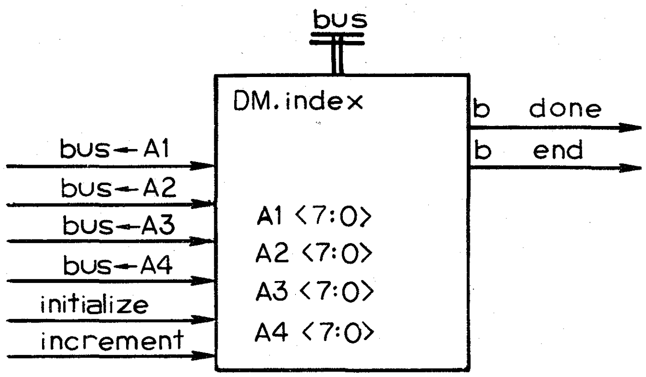


Figure 9—(a) DM.mult—multiply unit



(b) DM.index — FFT address generator. The DM.index control lines are described in Table IV

RTM LEVEL DESIGN

A block diagram of the processor structure is shown in Figure 7. It is a three bus system with each of the above operations performed on a separate bus. Figure 8 shows the specific RTM modules used; Table III describes the modules.

With the exception of DM.mult and DM.index, the data modules shown in Figure 8 are all standard RTM's. The functions of the two nonstandard modules are outlined below and illustrated in Figure 9.

TABLE IV—Description of Control Lines for Indexing Unit DM.Index

control line	function
initialize	initialize indexing unit
increment	calculate next 4 operand addresses for complex calculation
bus ← A1	load 1st address on bus
bus ← A2	load 2nd address on bus
bus ← A3	load 3rd address on bus
bus ← A4	load 4th address on bus
done	signals end of calculations involving one complex multiplier
-end	signals end of FFT

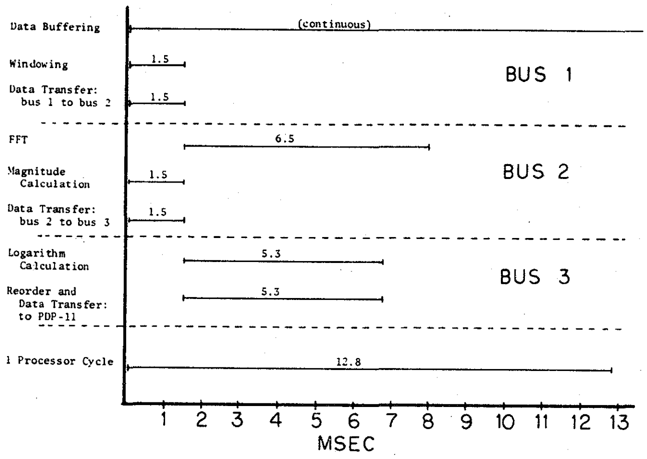


Figure 10—Processor timing diagram

DM.mult

This module multiplies the two 16 bit positive numbers in registers A and B. Any 16 bits of the 32 bit result can be placed on the bus. The multiplier was implemented using Fairchild 9344 2×4 bit multipliers.

DM.index

High speed hardware indexing units for FFT operand address generation have been presented in the literature.[8] This module generates the addresses of the four operands of every complex calculation during the FFT. It is a hardware implementation of the recursive equations for the FFT algorithm for real value inputs discussed previously. It was designed to sequence through all calculations involving one complex multiplier. Table IV defines the control lines shown in Figure 9(b).

The four 8-bit registers, A1, A2, A3 and A4 hold the addresses of the four operands. These registers do not physically exist since the addresses are generated combinatorily upon command; they are defined for logical purposes only.

Figure 10 shows the timing diagram of the processor. All arithmetic operations, register transfers, and memory accesses involve use of the bus, which has a settling time of 500 nsec. Therefore, the average speed of any operation is 500 nsec. This value was used in calculating the processing times shown in Figure 10. For example, approximately 13,000 operations are required to perform each 256 point FFT on bus 2. The processing time, therefore, is 6.5 msec. Bus 1 is continually buffering data, however, only 1.5 msec of 1 processor cycle (12.8 msec) are spent windowing 256 samples and transferring them to bus 2. Bus 2 spends 1.5 msec simultaneously accepting data from bus 1, calculating the magnitude of the harmonic components and transferring the results to bus 3. 6.5 msec are spent calculating the FFT. This leaves 4.8 msec (12.8-1.5-6.5) of dead-time during each processor cycle; time when no processing occurs on bus 2. Bus 3 spends 1.5 msec accepting data from bus 2, and 5.3 msec simultaneously calculating the logarithm of 129 samples and transferring them to the PDP-11. This leaves 6 msec of dead-time on bus 3. It is clear that bus 2 carries the heaviest processing load; therefore, bus 2 dead-time determines that a speed margin of 4.8 msec exists; that is, the processor completes processing each set of 256 samples 4.8 msec faster than needed to maintain real-time operation.

Accuracy

The question of accuracy always arises for a processo. operated in fixed point mode. As noted previously,[5] distribution of the $1/N$ normalization factor over the entire transform constrains the magnitudes of the operands at each level to prevent overflows. The only overflow possibility occurs during the calculation of the magnitude of the Fourier coefficients. When overflow occurs (positive or negative), the largest (positive or negative) number will be chosen.

Simulation runs to determine the effect of multiplier size on accuracy were conducted. A 16×16 bit multiplier was used in conjunction with the fixed point FFT described to process actual speech signal samples. For audible speech, accuracy of 1 percent relative mean square error was achieved when compared to floating point results. The same simulation using a 12×12 bit multiplier resulted in an error of 6 percent. For signals of small magnitude (such as the signal generated by silence) the error for the 16×16 bit multiplier rose to 25 percent; however, this is acceptable for processing the silence signal. For comparison, previous published accuracy results for a 16×16 bit multiplier and similar FFT algorithm[7] showed a maximum error of ±0.012 percent fullscale with a standard deviation of ±0.004 percent fullscale. On the basis of these results, the 12×12 bit multiplier was considered too inaccurate; therefore, the 16×16 bit multiplier was chosen.

RTM control

RTM control logic is designed with 2 basic modules:

1. Ke: a module which initiates arithmetic operations, data transfers between registers, and memory read/write cycles.

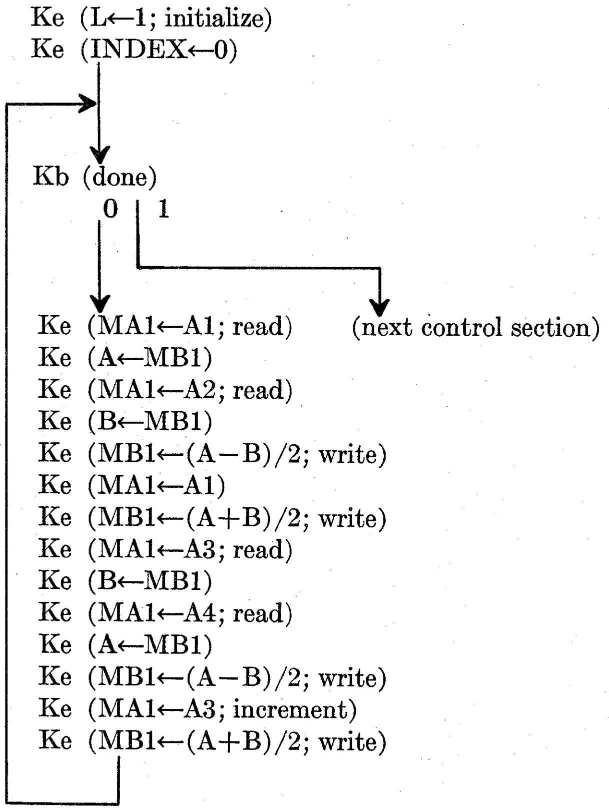
2. Kb: a module which chooses a control branch based on the value of a boolean flag.

With these modules the control for executing an algorithm can be specified in a manner quite similar to programming the algorithm in a high level programming language. This greatly simplifies the design of the control, thus resulting in a significant reduction in design time.

This concept can easily be illustrated by investigating a section of bus 2 control. This particular section controls the complex calculation for the degenerate case of W^0, that is, when the complex multiplier is $1+i0$. For this case the equations shown in Figure 5 reduce to

$$\begin{aligned} a' &= a+c \\ b' &= b+d \\ c' &= a-c \\ d' &= d-b \end{aligned}$$

A and B are general purpose arithmetic unit registers; INDEX is a storage register used for sequencing the counter through the 64 complex multipliers; ONE is a constant generator containing a "1"; and MA1 and MB1 are memory address and buffer registers, respectively. The control for this series of complex calculations is then:



```
Ke (L←1; initialize)
Ke (INDEX←0)
Kb (done)
   0 | 1
     |  (next control section)
Ke (MA1←A1; read)
Ke (A←MB1)
Ke (MA1←A2; read)
Ke (B←MB1)
Ke (MB1←(A−B)/2; write)
Ke (MA1←A1)
Ke (MB1←(A+B)/2; write)
Ke (MA1←A3; read)
Ke (B←MB1)
Ke (MA1←A4; read)
Ke (A←MB1)
Ke (MB1←(A−B)/2; write)
Ke (MA1←A3; increment)
Ke (MB1←(A+B)/2; write)
```

By dividing the results of each complex calculation by 2, the $1/N$ normalization factor can be distributed over the entire calculation.

The control section for the remaining complex calculations is, of course, more complex requiring 46 Ke and 7 Kb, but its design and implementation remain straightforward. To accomplish control of all operations on bus 2, including accepting data from bus 1, executing the FFT, calculating the magnitudes of the Fourier coefficients, and transferring data to bus 3, about 120 Ke and 20 Kb were used.

FUTURE EXTENSIONS

The speech processing application for this processor involves an initial Fourier transform, a second Fourier transform to obtain the cepstrum and an inverse Fourier transform. Figure 6 shows data flow for the proposed final form of the pipeline processor.

The present system is memory limited because 14 bus transfers in and out of memory are required for every complex calculation. Approximately 500 nsec are required for a bus transfer; 250 nsec to load data on the bus and 250 nsec to read data from the bus. Faster memory and bus systems can decrease this portion of the processing time. The processor fulfills both the overall goal of a modular FFT computer to meet the minimum processing rate of 10K data samples/sec, and attain accuracy of 1 percent relative mean square error necessary for speech analysis. This was done using existing RTM's with only 2 new modules required.

It should be emphasized that while the processor performs a specialized function (calculating the FFT), the RTM modules themselves, with the exception of DM.index, are general and can be used to implement any processor. In fact, since only the back plane wiring determines the characteristics of the processor, one set of RTM modules can be shared among many processors, if the processors will not be used simultaneously. This can result in substantial savings over the purchase or construction of several complete processors.

Along these lines, it would be advantageous to develop more complex but still general RTM modules. Specifically, a generalized micro-programmed LSI RTM module could be coded to implement the entire complex calculation, the FFT address generator, or any other algorithm on a single card. The complex calculation is an area where the system's speed can be significantly improved. At present, 46 bus transfers are required for each complex calculation. This number could be reduced by a factor of 3 by constructing one card to perform the entire complex calculation. The present system's specifications did not require such improvements and the RTM design concepts were used to investigate various system designs using existing modules rather than constructing an entire system from the start.

SUMMARY

This paper has reviewed the basic FFT algorithms and presented a method by which a relatively sophisticated piece of hardware such as an FFT processor could be designed at the register transfer level in a much shorter time than required in a conventional gate level design. The simplicity of this modular construction has permitted a fairly in-depth view of the processor. The resultant product and its method of implementation are rather unique in that they combine the convenience of a control logic that is similar in structure to software algorithms with the processing speed of a completely hard-wired algorithm.

ACKNOWLEDGMENTS

The authors wish to acknowledge the assistance of Lee Bitterman, and Professors Gordon Bell and Raj Reddy of CMU in the design and implementation of this FFT processor.

REFERENCES

1 C G BELL et al
The description and use of register transfer modules (RTM's)
IEEE Transactions on Computers Vol C-21 1972

2 R W SCHAFER L R RABINER
System for automatic formant analysis of voiced speech
The Journal of the Acoustical Society of America Vol 47 No 2 1970

3 E L HALL et al
Generation of products and quotients using approximate binary logarithms for digital filtering application
IEEE Transactions on Computers Vol C-19 1970

4 G D BERGLAND
A guided tour of the fast fourier transform
IEEE Spectrum Vol 6 1969

5 G D BERGLAND
A fast fourier transform algorithm for real valued series
Communications of the ACM Vol 11 1968

6 B GOLD et al
The FDP, a fast programmable signal processor
IEEE Transactions on Computers Vol C-20 No 1 1971

7 J W HARTWELL
A procedure for implementing the fast fourier transform on small computers
IBM Journal of Research and Development Vol 15 1971

8 W W MOYER
A high-speed indexing unit for FFT algorithm implementation
Computer Design Vol 10 No 12 1971

A systematic approach to the design of digital bussing structures*

by KENNETH J. THURBER, E. DOUGLAS JENSEN, and LARRY A. JACK

Honeywell, Inc.
St. Paul, Minnesota

and

LARRY L. KINNEY, PETER C. PATTON, and LYNN C. ANDERSON

University of Minnesota
Minneapolis, Minnesota

INTRODUCTION

Busses are vital elements of a digital system—they interconnect registers, functional modules, subsystems, and systems. As technological advances raise system complexity and connectivity, busses are being recognized as primary architectural resources which can frequently be the limiting factor in performance, modularity, and reliability. The traditional view of bussing as just an ad hoc way of hooking things together can no longer be relied upon to produce even viable much less cost-effective solutions to these increasingly sophisticated interconnect problems.

This paper formulates a more systematic approach by abstracting those bus parameters which are common to all levels of the system hierarchy. Every bus, whether it connects registers or processors, can be characterized by such factors as type and number, control method, communication mechanism, data transfer conventions, width, etc. Evaluating these parameters in terms of the preliminary functional requirements and specifications of the system constitutes an efficient procedure for the design of a cost-effective bus structure.

BUS STRUCTURE PARAMETERS

Each of these bus structure parameters involves a variety of interrelated tradeoffs, the most important of which are considered below.

Type and number of busses

Busses can be separated into two generic types: dedicated, and nondedicated.

Dedicated busses

A dedicated bus is permanently assigned to either one function or one physical pair of devices. For example, the Harvard class computer characterized by Figure 1 has two busses, each of which is dedicated according to both halves of the definition. One bus supplies procedure to the processor, the other provides data. If there were multiple procedure memory modules on the procedure bus, that bus would be functionally but not physically dedicated. The concept of "function" is hierarchical rather than atomic; in the sense that the procedure bus of Figure 1 carries both addresses and operands, it could be viewed as physically but not functionally dedicated. This dichotomy is reversed in Figure 2, which illustrates another form of Harvard class machine. In this case, one bus is functionally dedicated to addresses and the other to operands. They are undedicated from the standpoint of data/procedure separation, and physically undedicated as well.

The principal advantage of a dedicated bus is high throughput, because there is little, if any, bus contention (depending on the type and level of dedication). As a result, the bus controller can be quite simple compared to that of a non-dedicated bus. Also, portions of the communication mechanism which must be explicit in undedicated busses may be integral parts of the

* This work was supported in part by the Naval Air Development Center, Warminster, Pa., under Navy contract number N62269-72-C-0051.

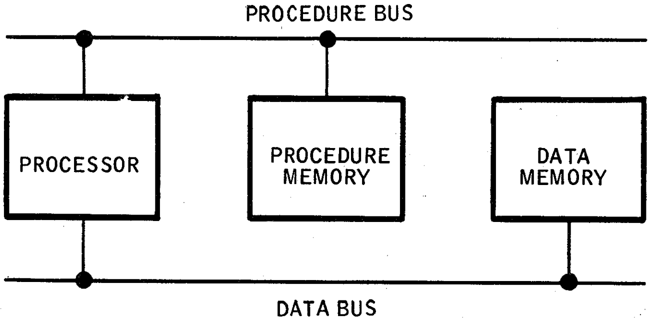


Figure 1—Harvard class computer with dedicated procedure and data busses

devices on a dedicated bus: addresses may be unnecessary, and the devices may automatically be in sync.

A system may include as many undedicated busses as its logical structure and data rates require, to the extreme of one or more busses between every pair of devices (Figure 3).

A major disadvantage of dedicated busses is the cost of the cables, connectors, drivers, etc., and of the multiple bus interfaces (although the interfaces are generally less complex than those for nondedicated busses). If reliability is a concern, the busses must be replicated to avoid potential single-point failures. Dedicated busses do not often support system modularity, because to add a device frequently involves adding new interfaces and cables.

Non-dedicated busses

Non-dedicated busses are shared by multiple functions and/or devices. As pointed out earlier, busses may be functionally dedicated and physically non-dedicated, or vice versa. The Princeton class computer of Figure 4 illustrates a commonly encountered type of single bus structure which is not dedicated on either a functional or a physical basis. The interesting case of multiple, system-wide, functionally and physically non-dedicated busses is seen in Figure 5. Here every device can communicate with every other device using any bus, so the failure of a bus interface to some device simply reduces the number of busses (but not devices) remaining available to that device.

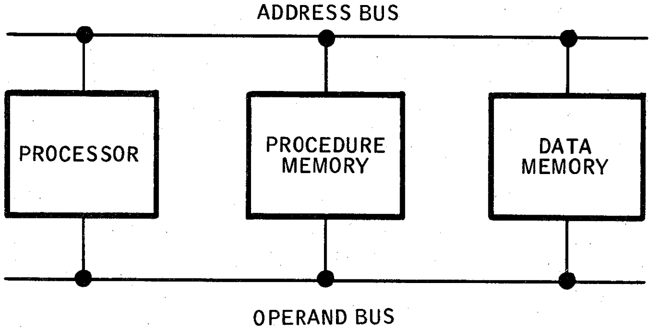


Figure 2—Harvard class computer with dedicated address and operand busses

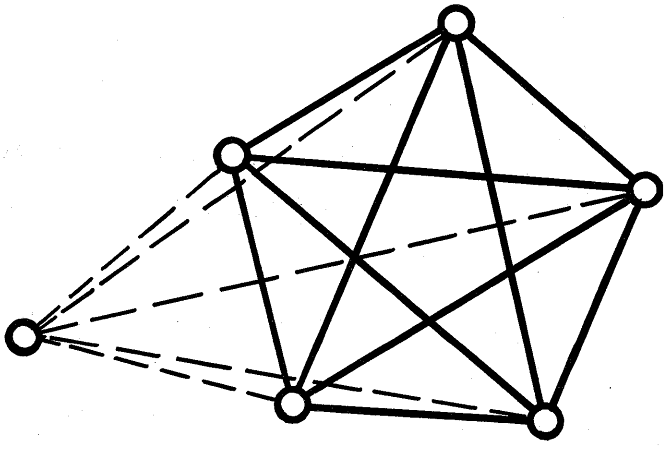


Figure 3—Adding a device to a non-dedicated bus structure

The crossbar matrix is a form of non-dedicated bus structure for connecting any element of one device class (such as memories) to any element of another (such as processors). It can be less efficiently used to achieve complete connectivity between all system devices. The crossbar can be very complex to control, and the number of switches increases as the square of the number of devices, as shown in Figure 6. It also suffers from the disability that failure of a crosspoint leaves no alternative path between the corresponding devices.

By adding even more hardware, the crossbar switch can be generalized to a code-activated network (analogous to the telephone system) in which devices seek their own paths to each other.

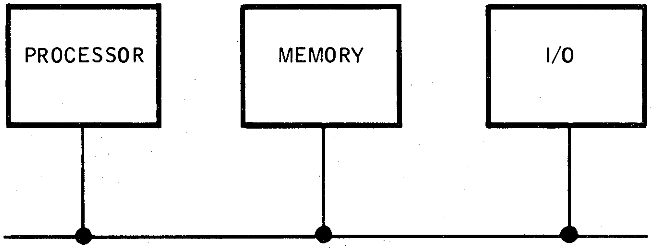


Figure 4—Princeton class computer with a single non-dedicated bus

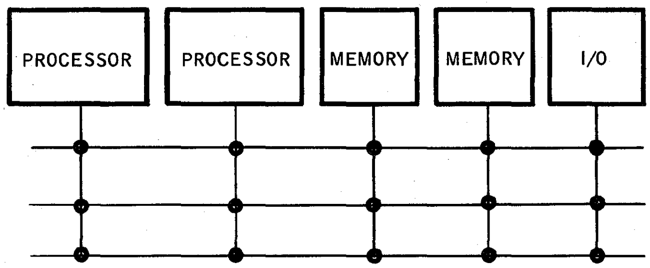


Figure 5—Multiple, system-wide, non-dedicated busses

Another relatively unconventional non-dedicated bus structure is the permutation or sorting network which can connect N devices to N other devices. The sorting network may be implemented with memory or gating, but in either case if all N! permutations are allowed, the hardware is extensive for anything but very small N's.

Non-dedicated busses offer modularity as their main advantage, in that devices generally may be added to them more easily than to dedicated busses. Multiple busses such as those in Figure 5 not only increase bandwidth but also enhance reliability, rendering the system fail-soft. While non-dedicated busses avoid the proliferation of cables, connectors, drivers, etc., they do exact a toll in usage conflict. Bus allocation requires logic and time, and if this time cannot be masked by data transfers, the bus bandwidth and/or assignment algorithm may have to be compromised. Furthermore, the devices which desire but do not obtain the bus must wait for another opportunity to contend for it.

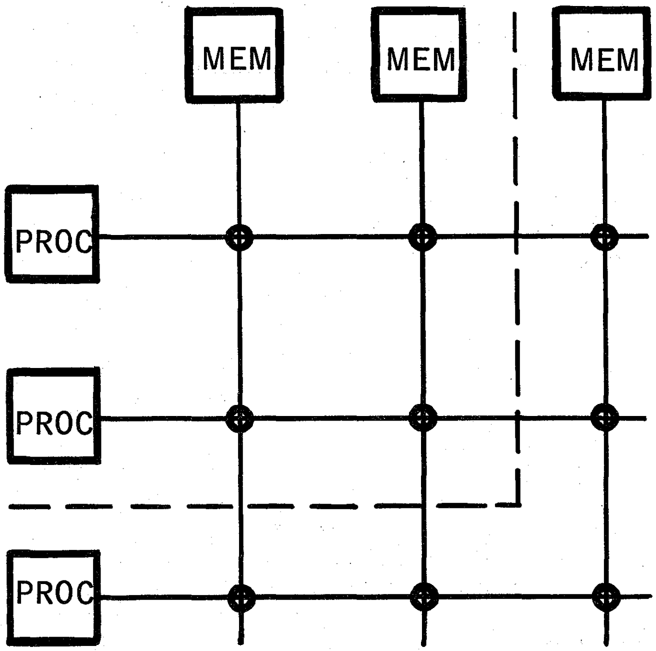


Figure 6—Adding devices to a crossbar bus

The communication technique is usually more complex for non-dedicated busses, because devices must be explicitly addressed and synchronized.

Bus control techniques

When a bus is shared by multiple devices, there must be some method whereby a particular unit requests and obtains control of the bus and is allowed to transmit data over it. The major problem in this area is resolution of bus request conflicts so that only one unit obtains the bus at a given time. The different control schemes can be roughly classified as being either centralized or decentralized. If the hardware used for passing bus control from one device to another is largely concentrated in one location, it is referred to as centralized control. The location of the hardware could be within one of the devices which is connected to the bus, or it could be a separate hardware unit. On the other hand, if the bus control logic is largely distributed throughout the different devices connected to the bus, it is called decentralized control.

The various bus control techniques will be described here in terms of distinct control lines, but in most cases the equivalent functions can be performed with coded transfers on the bus data lines. The basic tradeoff is allocation speed versus total number of bus lines.

Centralized bus control

With centralized control, a single hardware unit is used to recognize and grant requests for the use of the bus. At least three different schemes can be used, plus various modifications or combinations of these:

- Daisy Chaining
- Polling
- Independent Requests.

Centralized Daisy Chaining is illustrated in Figure 7.

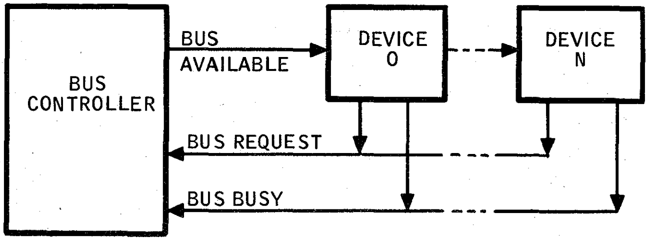


Figure 7—Centralized bus control: daisy chain

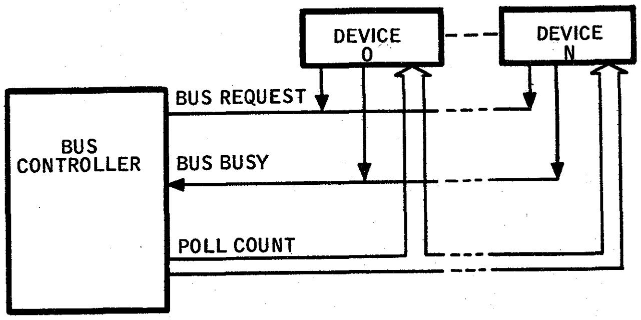


Figure 8a—Centralized bus control: polling with a global counters

Each device can generate a request via the common Bus Request line. Whenever the Bus Controller receives a request on the Bus Request line, it returns a signal on the Bus Available line. The Bus Available line is daisy chained through each device. If a device receives the Bus Available signal and does not want control of the bus, it passes the Bus Available signal on to the next device. If a device receives the Bus Available signal and is requesting control of the bus, then the Bus Available signal is not passed on to the next device. The requesting device places a signal on the Bus Busy line, drops its bus request, and begins its data transmission. The Bus Busy line keeps the Bus Available line up while the transmission takes place. When the device drops the Bus Busy signal, the Bus Available line is lowered. If the Bus Request line is again up, the allocation procedure repeates.

The Bus Busy line can be eliminated, but this essentially converts the bus control to a decentralized Daisy Chain (as discussed later).

The obvious advantage of such a scheme is its simplicity: very few control lines are required, and the number of them is independent of the number of devices; hence, additional devices can be added by simply connecting them to the bus.

A disadvantage of the Daisy Chaining scheme is its susceptibility to failure. If a failure occurs in the Bus Available circuitry of a device, it could prevent succeeding devices from ever getting control of the bus or it could allow more than one device to transmit over the bus at the same time. However, the logic involved is quite simple and could easily be made redundant to increase its reliability. A power failure in a single device or the necessity to take a device off-line can also be problems with the Daisy Chain method of control.

Another disadvantage is the fixed priority structure which results. The devices which are "closer" to the Bus Controller always receive control of the bus in preference to those which are "further away". If the closer devices had a high demand for the bus, the further devices could be locked out.

Since the Bus Available signal must sequentially ripple through the devices, this bus assignment mechanism can also be quite slow.

Finally, it should be noted that with Daisy Chaining, cable lengths are a function of system layout, so adding, deleting, or moving devices is physically awkward.

Figure 8a illustrates a centralized Polling system. As in the centralized Daisy Chaining method, each device on the bus can place a signal on the Bus Request line. When the Bus Controller receives a request, it begins polling the devices to determine who is making the request. The polling is done by counting on the polling lines. When the count corresponds to a requesting device, that device raises the Bus Busy line. The controller then stops the polling until the device has completed its transmission and removed the busy signal. If there is another bus request, the count may restart from zero or may be continued from where it stopped.

Restarting from zero each time establishes the same sort of device priority as proximity does in Daisy Chaining, while continuing from the stopping point is a round-robin approach which gives equal opportunity to all devices. The priorities need not be fixed because the polling sequence is easily altered.

The Bus Request line can be eliminated by allowing the polling counter to continuously cycle except while it is stopped by a device using the bus. This alternative impacts the restart (i.e., priority) philosophy, and the average bus assignment time.

Polling does not suffer from the reliability or physical placement problems of Daisy Chaining, but the number of devices in Figure 8a limited by the number of polling lines. Attempting to poll bit-serially involves synchronous communication techniques (as described later) and the attendant complications.

Figure 8b shows that centralized Polling may be made independent of the number of devices by placing a counter in each device. The Bus Controller then is reduced to distributing clock pulses which are counted

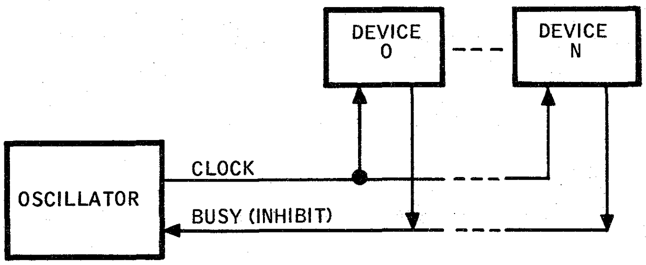


Figure 8b—Centralized bus control: polling with local counters

by all devices. When the count reaches the code of a device wanting the bus, the device raises the Busy line which inhibits the clock. When the device completes its transmission, it removes the Busy signal and the counting continues. The devices can be serviced either in a round-robin manner or on a priority basis. If the counting always continues cyclically when the Busy signal is removed, the allocation is round-robin, and if the counters are all reset when the Busy signal is removed, the devices are prioritized by their codes. It is also possible to make the priorities adaptive by altering the codes assigned to the devices. The clock skew problems tend to limit this technique to small slow systems; it is also exceptionally susceptible to noise and clock failure.

Polling and Daisy Chaining can be combined into schemes where addresses or priorities are propagated between devices instead of a Bus Available signal. This adds some priority flexibility to Daisy Chaining at the expense of more lines and logic.

The third method of centralized bus control, Independent Requests, is shown in Figure 9. In this case each device has a separate pair of Bus Request and Bus Granted lines, which it uses for communicating with the Bus Controller. When a device requires use of the bus, it sends its Bus Request to the controller. The controller selects the next device to receive service and sends a Bus Granted to it. The selected device lowers its request and raises Bus Assigned, indicating to all other devices that the bus is busy. After the transmission is complete, the device lowers the Bus Assigned line and the Bus Controller removes Bus Granted and selects the next requesting device.

The overhead time required for allocating the bus can be shorter than for Daisy Chaining or Polling since all Bus Requests are presented simultaneously to the Bus Controller. In addition, there is complete flexibility available for selecting the next device for service. The controller can use prespecified or adaptive priorities, a round-robin scheme, or both. It is also possible to disable requests from a particular device which, for instance, is known or suspected to have failed.

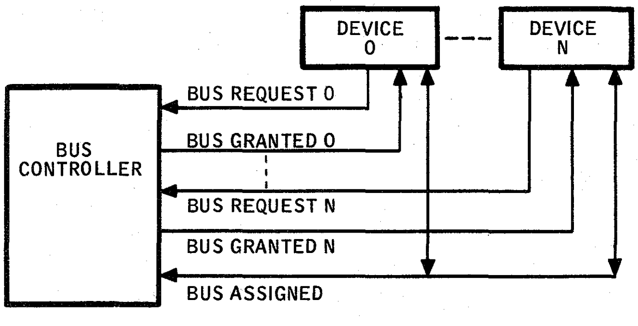


Figure 9—Centralized bus control: independent requests

The major disadvantage of Independent Requests is the number of lines and connectors required for control. Of course, the complexity of the allocation algorithm will be reflected in the amount of Bus Controller hardware.

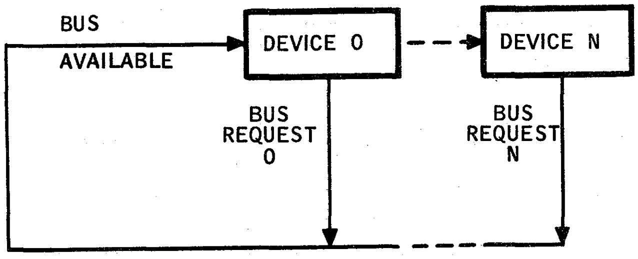


Figure 10a—Decentralized bus control: daisy chain 1

Decentralized bus control

In a decentrally controlled system, the control logic is (primarily) distributed throughout the devices on the bus. As in the centralized case, there are at least three distinct schemes, plus combinations and modifications of these:

- Daisy Chaining
- Polling.
- Independent Requests

A decentralized Daisy Chain can be constructed from a centralized one by omitting the Bus Busy line and connecting the common Bus Request to the "first" Bus Available, as shown in Figure 10a. A device requests service by raising its Bus Request line if the incoming Bus Available line is low. When a Bus Available signal is received, a device which is not requesting the bus passes the signal on. The first device which is requesting service does not propagate the Bus Available, and keeps its Bus Request up until finished with the bus. Lowering the Bus Request lowers Bus Available if no successive devices also have Bus Request signals up, in which case the "first" device wanting the bus gets it. On the other hand, if some device "beyond" this one has a Bus Request, control propagates down to it. Thus, allocation is always on a round-robin basis.

A potential problem exists in that if a device in the interior of the chain releases the bus and no other device is requesting it, the fall of Bus Request is propagating back toward the "first" device while the Bus Available signal propagates "forward." If devices on both

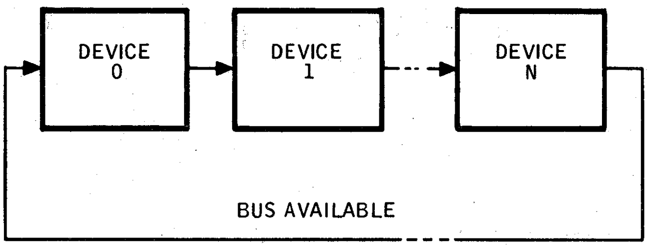


Figure 10b—Decentralized bus control: daisy chain 2

sides of the last user now raise Bus Request, the one to the "right" will obtain the bus momentarily until its Bus Available drops when the "left" device gets control. This dilemma can be avoided by postponing the bus assignment until such races have settled out, either asynchronously with one-shots in each device or with a synchronizing signal from elsewhere in the system.

A topologically simpler decentralized Daisy Chain is illustrated in Figure 10b. Here, it is not possible to unambiguously specify the status of the bus by using a static level on the Bus Available line. However, it is possible to determine the bus status from transitions on the Bus Available line. Whenever the Bus Available coming into a device changes state and that device needs to use the bus, it does not pass a signal transition on to the next device; if the device does not need the bus, it then changes the Bus Available signal to the next device. When the bus is idle, the Bus Available signal oscillates around the Daisy Chain. The first device to request the bus and receive a Bus Available signal change terminates the oscillation and takes control of the bus. When the device is finished with the bus, it causes a transition in Bus Available to the next device.

Dependence on signal edges rather than levels renders this approach somewhat more susceptible to noise than the previous one. This problem can be minimized by passing control with a request/acknowledge type of mechanism such as described later for communication, although this slows bus allocation. Both of these decentralized Daisy Chains have the same single-point failure mode and physical layout liabilities as the centralized version. Specific systems may prefer either the (centralized) priority or the (decentralized) round-robin algorithm, but they are equally inflexible (albeit simple).

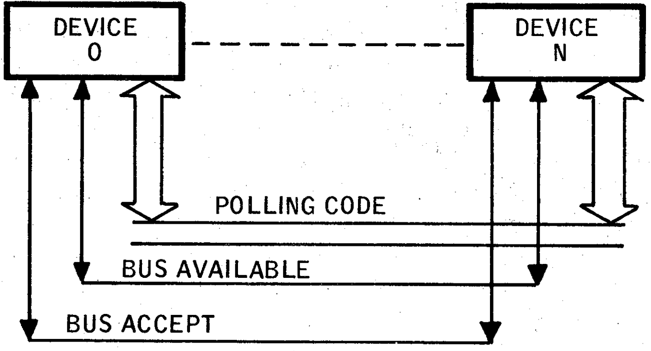


Figure 11—Decentralized bus control: polling

Decentralized Polling can be performed as shown in Figure 11. When a device is willing to relinquish control of the bus, it puts a code (address or priority) on the polling lines and raises Bus Available. If the code matches that of another device which desires the bus, that device responds with Bus Accept. The former device drops the polling and Bus Available lines, and the latter device lowers Bus Accept and begins using the bus. If the polling device does not receive a Bus Accept (a Bus Refused line could be added to distinguish between devices which do not desire the bus and those which are failed), it changes the code according to some allocation algorithm (round-robin or priority) and tries again. This approach requires that exactly one device be granted bus control when the system is initialized. Since every device must have the same allocation hardware as a centralized polling Bus Controller, the decentralized version utilizes substantially more hardware. This buys enhanced reliability in that failure of a single device does not necessarily affect operation of the bus.

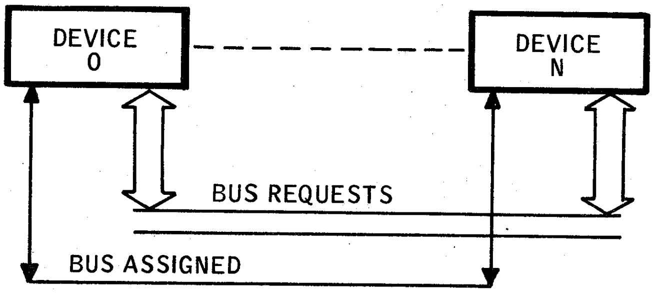


Figure 12—Decentralized bus control: independent requests

Figure 12 illustrates the decentralized version of Independent Requests. Any device desiring the bus raises its Bus Request line, which corresponds to its priority. When the current user releases the bus by dropping Bus Assigned, all requesting devices examine all active Bus Requests. The device which recognizes itself as the highest priority requestor obtains control of the bus by raising Bus Assigned. This causes all other requesting devices to lower their Bus Requests (and to store the priority of the successful device if a round-robin algorithm is to be accomodated).

The priority logic in each device is simpler than that in the centralized counterpart, but the number of lines and connectors is higher. If the priorities are fixed rather than dynamic, not all request lines go to all devices, so the decentralized case uses fewer lines in systems with up to about 10 devices. Again, the decentralized method offers some reliability advantages over the centralized one.

The clock skew problems limit this process to small dense systems, and it is exceptionally susceptible to noise and clock failure.

Bus communication techniques

Once a device has obtained control of a bus, it must establish contact with the desired destination. The information required to do this includes

- Source Address
- Destination Address
- Communication Class
- Action Class.

The source address is often implicit, and the destination address may be also, in the case of a dedicated bus. Communication class refers to the type of information to be transferred: e.g., data, command, status, interrupt etc. This too might be partially or wholly implicit, or might be merged with the action class, which determines the function to be performed, such as input, output, etc. After this initial coordination has been accomplished, the actual communication can proceed. Information may be transferred between devices synchronously, asynchronously, or semisynchronously.

Synchronous bus communication

Synchronous transmission techniques are well understood and widely used in communication systems, primarily because they can efficiently operate over long lengths of cable. A synchronous bus is characterized by the existence of fixed, equal-width time slots which are either generated or synchronized by a central timing mechanism.

The bus timing can be generated globally or both globally and locally. A globally timed bus contains a central oscillator which broadcasts clock signals to all units on the bus. Depending on the logical structure and physical layout of the bus, clock skew may be a serious problem. This can be somewhat alleviated by distributing a globally generated frame signal which synchronizes a local clock in each device. The local clocks drive counters which are decoded to identify the time slot assigned to each device. A sync pulse occurs every time the count cycle (i.e., frame) restarts. The device clocks must be within the initial frequency and temperature coefficient tolerances determined by the bus timing characteristics. Skew can still exist if a separate frame sync line is used, but can be avoided by putting frame sync in the data. The sync signal then must be separable from the data, generally through amplitude, phase, or coding characteristics. If the identifying characteristic is amplitude, the line drivers and receivers are much more complex analog circuits than those for simple binary data. If phase is used, the sync signal must be longer than a time slot, which costs bus bandwidth and again adds an analog dimension to the drivers and receivers. If the sync signal is coded as a special binary sequence, it could be confused with normal data, and can require complex decoders.

All of the global and global/local synchronization techniques are quite subject to noise errors.

There are two basic approaches to synchronous busses: the time slots may be assigned to devices on either a dedicated or non-dedicated basis. A mix of both dedicated and undedicated slots can also be used. If time slots are dedicated, they are permanently allocated to a device regardless of how frequently or infrequently that device uses them. Each device on the bus is allowed to communicate on a rotational (time division multiplex) basis. The only way that any priority can be established is by assigning more than one slot to a device (sometimes call super-commutation). More than one device may be assigned to a single time slot by submultiplexing (subcommutating) slower or mutually exclusive devices.

Generally, not all devices will wish to transmit at once; system requirements may not even require or permit it. If any expansion facilities for additional devices are provided, many of the devices may not even be implemented on any given system. These two factors tend to waste bus bandwidth, and lowering the bandwidth to an expected "average" load may risk unacceptable conflicts and delays in peak traffic periods.

Another difficulty that reduces throughput on a dedicaded time slot bus is that devices frequently are not all the same speed. This means that if a device operates slower than the time slot rate, it cannot run at its full speed. The time slot rate could be selected to match the rate of the slowest device on the bus, but this slows down all faster devices. Alternatively, the time slot rate can be made as fast as the fastest device on the bus, and buffers incorporated into the slower devices. Depending on the device rate mismatches and the length of data blocks, these buffers could grow quite large. In addition, the buffers must be capable of simultaneous input and output (or one read and one write in a time slot period), or else the whole transfer is delayed until the buffer is filled. Another approach is to run the bus slower than the fastest device and assign multiple time slots to that device, which complicates the control and wastes bus bandwidth if that device is not always transferring data. Special logic must also be included if burst or block transfers are to be permitted, since a device normally does not get adjacent time slots.

For reliability, it is generally desirable that the receiving device verify and acknowledge correct arrival of the data. This is most effectively done on a word basis unless the physical nature of the transmitting device precludes retry on anything other than a complete block or message. If a synchronous time slot is wide enough to allow a reply for every word, then data transmission will be slower than with an asynchronous bus because the time slots would have to be defined by the slowest device on the bus. One solution is to establish a system convention that verification is by default, and if an error does occur, a signal will be returned to the source device N (say two) time slots later. The destination has time to do the validity test without slowing the transfer rate; however, the source must retain all words which have been transmitted but not verified.

Non-dedicated time slots are provided to devices only as needed, which improves bus utilization efficiency at the cost of slot allocation hardware. Block transfers and priority assignment schemes are possible if the bus assignment mechanism is fast enough. The device speed and error checking limitations of the dedicated case are also shared by non-dedicated systems.

Asynchronous bus communication

Asynchronous bus communication techniques fall into two general categories: One-Way Command, and Request/Acknowledge. A third case is where clocking information is derived from the data itself at the destination (using phase modulation, etc.); this is not treated here because it is primarily suited to long-distance bit-serial communications applications and is well documented elsewhere.

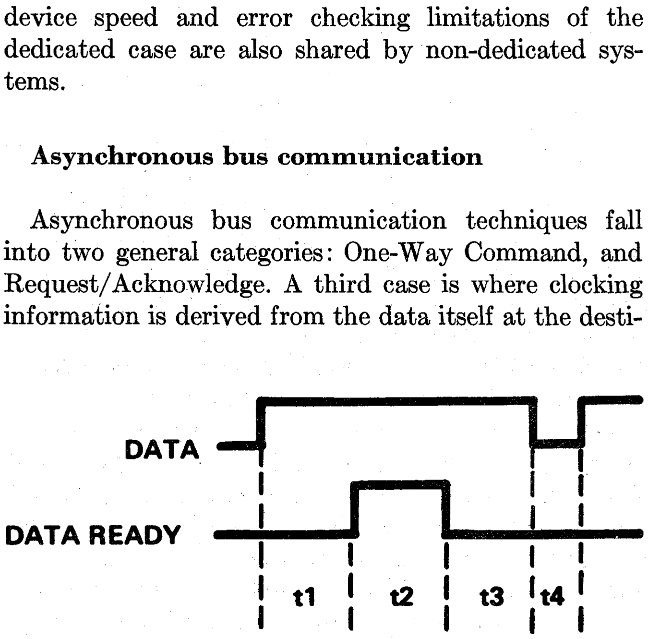


Figure 13—Asynchronous, source-controlled, one-way command communication

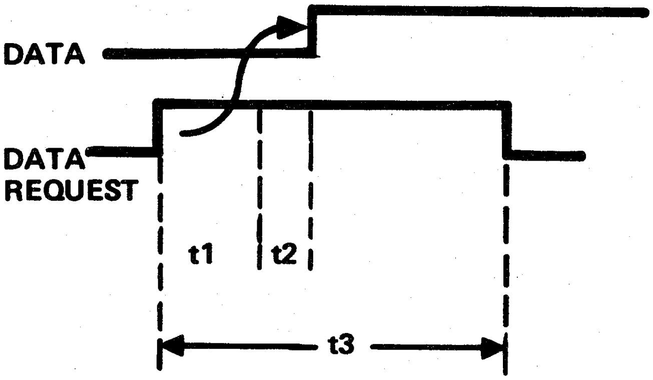


Figure 14—Asynchronous, destination-controlled, one-way command communication

One-Way Command refers to the fact that the data transfer mechanism is completely controlled by only one of the two devices communicating—once the transfer is initiated, there is no interaction (except, perhaps, for an error signal).

A One-Way Command (OWC) interface may be controlled by either the source or the destination device.

With a source-controlled OWC interface, the transmitting device places data on the bus, and signals Data Ready to the receiving device, as seen in Figure 13. Timing of Data Ready is highly dependent on implementation details, such as exactly how it is used by the destination device. If Data Ready itself directly strobes in the data, then it must be delayed long enough (t_1) for the data to have propagated down the bus and settled at the receiving end before Data Ready arrives. Instead of "pipelining" data and Data Ready, it is safer to allow the data to reach the destination before generating Data Ready, but this makes the transfer rate a function of the physical distance between devices. A better approach is to make Data Ready as wide as the data (i.e., $t_1 = t_3 = 0$), and let the receiving device internally delay before loading. t_4 is the time required either for the source device to reload its output data register, or for control of the bus to be reassigned.

The principal advantages of the source-controlled OWC interface are simplicity and speed. The major disadvantages are that there is no validity verification from the destination, it is difficult and inefficient to communicate between devices of different speeds, and noise pulses on the Data Ready line might be mistaken for valid signals. The noise problem can be minimized by proper timing, but usually at the expense of transfer rate.

The validity check problem can be avoided with a destination-controlled OWC interface, such as shown in Figure 14. The receiving device raises Data Request, which causes the source to place data on the bus. The destination now has the problem of deciding when to look at the data lines, which is related to the physical distance involved. If an error is detected in the word, the receiving device sends a Data Error signal instead of another Data Request, so the validity check time may limit the transfer rate. The speed is also adversely affected by higher initial overhead, and by twice the number of bus propagation delays as used by the source-controlled interface.

The Request/Acknowledge method of asynchronous communication can be separated into three cases: Non-Interlocked, Half-Interlocked, and Fully-Interlocked.

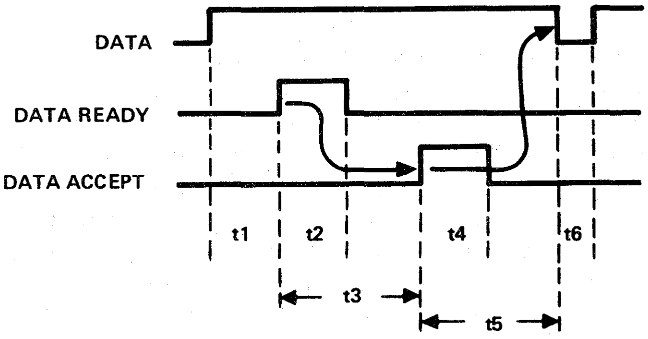


Figure 15—Asynchronous, non-interlocked, request/acknowledge communication

Figure 15 illustrates the Non-Interlocked method. The source puts data on the bus, and raises Data Ready; the destination stores the data and responds with Data Accept, which causes Data Ready to fall and new data to be placed on the lines. If an error is found in the data, the receiving device raises Data Error instead of Data Accept. This signal interchange not only provides error control, but also permits operation between devices of any speeds. The price is primarily speed, although some added logic is also required. As with the One-Way Command interface, the exact timing is a function of the implementation. There are now two lines susceptible to noise, and twice as many bus delays to consider. Improper ratios of bus propagation time and communication signal pulse widths could allow another Data Ready to come and go while Data Accept is still high in response to a previous one, which would hang up the entire bus.

This can be avoided by making Data Ready remain up until Data Accept (or Data Error) is received by the source, as seen in Figure 16. In this Half-Interlocked interface, if Data Ready comes up while Data Accept is still high, the transfer will only be delayed. Furthermore, the variable width of Data Ready tends to protect it from noise. There is no speed penalty and very little hardware cost associated with these improvements over the Non-Interlocked case.

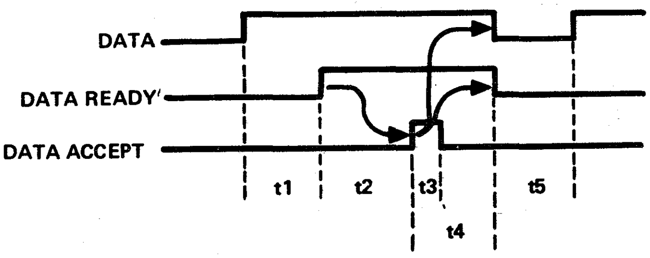


Figure 16—Asynchronous, half-interlocked, request/acknowledge communication

One more potential timing error is possible if Data Accept extends over the source buffer reload period and masks the leading edge of the next Data Ready. Figure 17 shows how this is avoided with a Fully-Interlocked interface where a new Data Ready does not occur until the trailing edge of the old Data Accept (or Data Error). Also, both communication signals are now comparatively noise-immune. The device logic is again slightly more complex, but the major disadvantage is that the bus delays have doubled over the Half-Interlocked case, nearly halving the transfer rate upper limit.

Semisynchronous bus communication

Semisynchronous busses may be thought of as having time slots which are not necessarily fixed equal width. On the other hand, they might also be viewed as essentially asynchronous busses which behave synchronously when not in use.

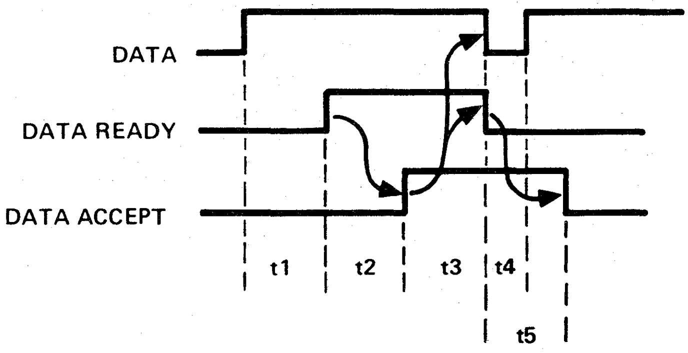


Figure 17—Asynchronous, fully-interlocked, request/acknowledge communication

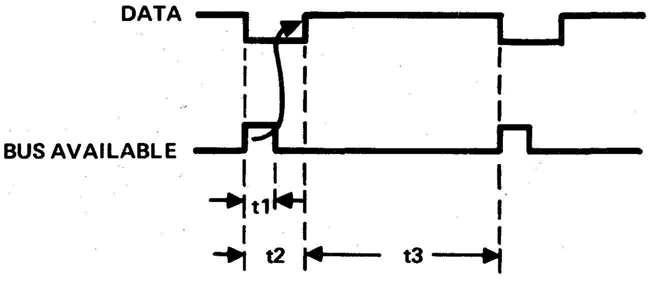


Figure 18—Semisynchronous, source-controlled, one-way command communication

Semisynchronous busses were devised to retain the basic asynchronous advantage of communication between different speed devices, while overcoming the asynchronous disadvantage of real-time error response and the synchronous disadvantage of clock skew. Error control in a synchronous system does not impede the transfer rate because the error signal can be deferred as many time slots as the validity test requires. This is not possible on a conventional asynchronous bus since there is no global timing signal available to all devices. Actually, this is true only when the bus is idle, because while it is in use there are one or more communication signals which may be observed by all devices. So an asynchronous bus could defer the Data Error signal for some N word-times as defined by whatever transfer technique is employed. But when no device is using the bus, these signals normally stop, so the one or more pairs of devices which transferred the last N words have no time reference for a deferred error response. The semisynchronous bus handles this problem by generating extra communication signals which serve as pseudoclock pulses for this purpose when the bus is idle. Only N pulses are actually needed, but a continuous oscillation may facilitate the restart of normal bus operation.

The location of this pseudoclock depends on the bus control method. If the bus is centrally controlled, the Bus Controller can detect the idle bus condition and generate the pseudoclock signals. A decentrally controlled bus requires that this function be performed by the last device to use the bus. The replication of logic adds cost, and if this last device should fail while generating the pseudoclocks, the entire bus will be down.

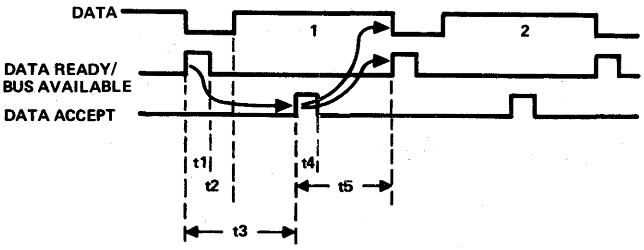


Figure 19—Semisynchronous, non-interlocked request/acknowledge communication (Data Ready/Bus Available)

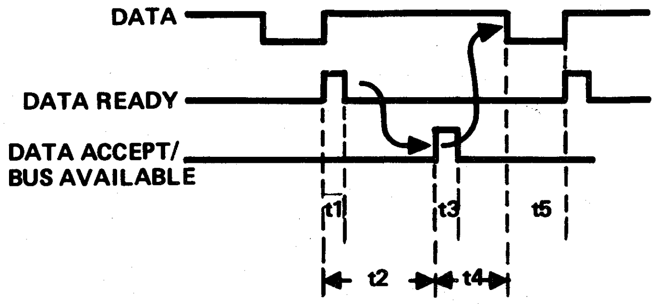


Figure 20—Semisynchronous, non-interlocked, request/acknowledge communication (Data Accept/Bus Available)

Like asynchronous busses, semisynchronous busses may be either One-Way Command or Request/Acknowledge.

Figure 18 illustrates how the timing of a semisynchronous source-controlled bus resembles that of its asynchronous counterpart (there is no corresponding destination-controlled case). Instead of the source device sending a Data Ready to signal the presence of new data, it sends a Bus Available to define the end of its time slot and the beginning of the next. During a time slot, the bus assignment for the following slot is made; Bus Available then causes the next device to place its destination address and data on the bus. The selected destination then waits for the data to settle, loads it, and generates another Bus Available.

Combining the function of Data Ready with that of Bus Available (a line generally required by an asynchronous bus) is a benefit which accrues to all semisynchronous busses. The semisynchronous One-Way Command interface does avoid the real-time error response, but it is still highly susceptible to noise, and incompatible with devices of differing speeds.

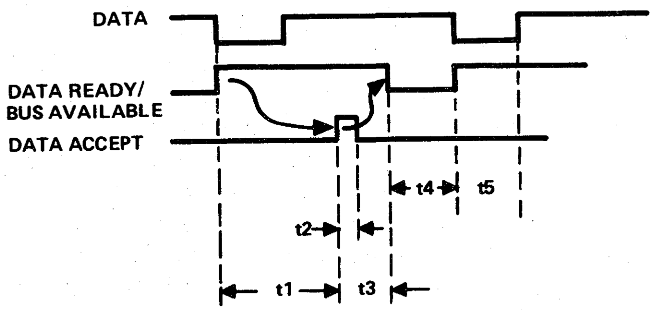


Figure 21—Semisynchronous, half-interlocked, request/acknowledge communication (Data Ready/Bus Available)

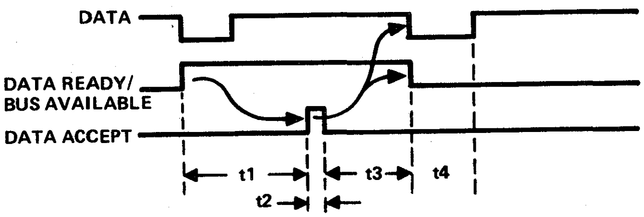


Figure 22—Semisynchronous, half-interlocked, request/acknowledge communication (toggling Data Ready/Bus Available)

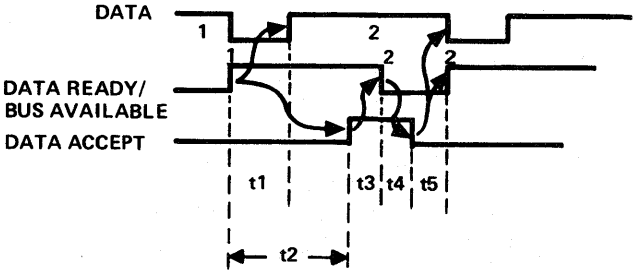


Figure 24—Semisynchronous, fully-interlocked, request/acknowledge communication (Data Ready/Bus Available)

For semisynchronous as well as asynchronous busses, there are Non-Interlocked, Half-Interlocked, and Fully-Interlocked Request/Acknowledge interfaces.

The Non-Interlocked interface shown in Figure 19 is a direct extension of the One-Way Command case. It handles devices of different speeds, but also is susceptible to noise and potential hangup.

However, a semisynchronous bus using Data Ready as Bus Available for a Non-Interlocked interface picks up one of the liabilities of synchronous busses. The transmitting device will not generate Bus Available until Data Accept has been received and its word-time is finished, which wastes bus bandwidth if a slower source is followed by a faster one in the next time slot. This can only be alleviated with the same sort of bus bandwidth and buffer size trade-offs that a synchronous bus would use to match different device speeds.

Figure 20 illustrates a scheme which solves this difficulty by using Data Accept for Bus Available. This optimizes bus bandwidth in the asynchronous sense that the transfer rate is slaved to the speed of the receiving device. Of course, the noise and hangup problems are still present.

Since using Data Ready as Bus Available is unsuccessful for Non-Interlocked interfaces, it is not surprising that it doesn't work in the Half-Interlocked case either. As seen in Figure 21, t_5 is wasted because only the leading edge of Data Ready is used as Bus Available. Also, one device would try to hold Bus Available up while another is pulling it down. The second device could wait for the first to release the line, but skew on the Data Accept line from the first destination to the first and second sources would cause the wait to be quite lengthy. Furthermore, if Data Accept must be used by both source devices, it may as well transfer control instead of Bus Available.

To keep from wasting t_5, it might be proposed that Bus Available simply be toggled and both edges be utilized as in Figure 22, but the same state conflict exists here. Toggling a Bus Available flip-flop with Data Accept makes no more sense than both source devices employing Data Accept, and would add time.

Thus, Data Accept must be converted to Bus Available, as shown in Figure 23. Except for a deferred error signal, the disabilities of a conventional Half-Interlocked asynchronous bus continue to apply.

The same reasoning causes the Fully-Interlocked interface of Figure 24 to be rejected for that of Figure 25, where the trailing edge of Data Accept serves as Bus Available.

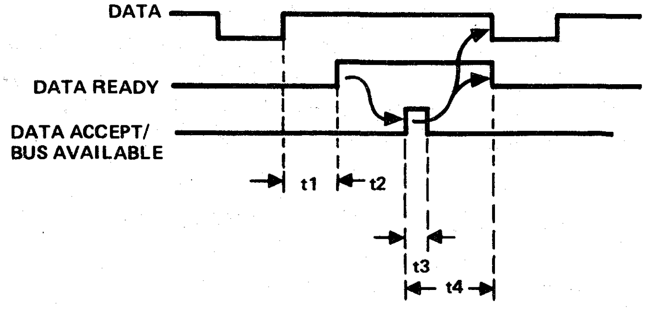


Figure 23—Semisynchronous, half-interlocked, request/acknowledge communication (Data Accept/Bus Available)

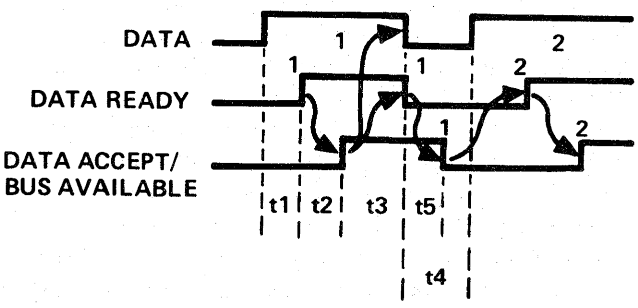


Figure 25—Semisynchronous, fully-interlocked, request/acknowledge communication (Data Accept/Bus Available)

Data transfer philosophies

There are five basic data transfer philosophies that can be considered for a bus:

- Single word transfers only
- Fixed length block transfers only
- Variable length block transfers only
- Single word or fixed length block transfers
- Single word or variable length block transfers.

(It should be noted that here the term "word" is used functionally to denote the basic information unit on the bus; bus width factors are covered later.)

The data transfer philosophy is directly involved with three other major aspects of the system: the access characteristics of the devices using the bus; the control mechanism by which the bus is allocated (if it is non-dedicated); and the bus communication techniques. Of course, if the bus connects functional units of a computer such as processors and memories, the data transfer philosophy may severely impact programming, memory allocation and utilization, etc.

Single words only

The choice of allowing only single words to be transferred has a number of important system ramifications. First, it precludes any effective use of purely block-oriented devices, such as disks, drums, or BORAMs. These devices have a high latency and their principal value lies in amortizing this time across many words in a block. To a lesser extent, this concern also applies to other types of devices. There can be substantial initial overhead in obtaining access to a device: bus acquisition, bus propagation, busy device delay, priority resolution, address mapping, intrinsic device access time, etc. Prorating these against a block of words would reduce the effective access time.

The second factor in a single-word-only system is the bus control method. Since a non-dedicated bus must be reassigned to another device for each word, the allocation algorithm may have to be very fast to meet the bus throughput specs. Even if bus assignment occurs in parallel with data transfer, this could restrict the sophistication of the algorithm, the bus bandwidth, or both. Judiciously selected parameters (speed, priorities, etc.) conceivably could enable a bus controller to handle blocks from a slow device on a word-by-word basis.

A single-word-only bus requires that the communication scheme operate at the word rate, whereas with block transfers it might be possible for devices to effect higher throughput by interchanging communication signals only at the beginning and end of each block.

Fixed length blocks only

Bus bandwidth may be increased at the expense of flexibility by transferring only fixed length blocks of data. Problems arise when the bus block size does not match that of a block-oriented device on the bus. If the bus blocks are smaller, some improvement is achieved over the single-word-only bus, but not as much as would be possible. If the bus blocks are too large, extraneous data is transferred, which wastes bus bandwidth and buffer space, and unnecessarily ties up both devices. However, there are applications such as lookaside memories where locality of procedure and data references make effective use of a purely fixed length block transfer philosophy.

Since the bus is assigned for entire blocks, the control can be slower and thus simpler. Likewise, the communication validity check can be restricted to blocks because this is the smallest unit that could be retried in case of an error. The Data Ready aspect of communication would have to remain on a word basis unless a self-clocked modulation scheme is used.

Variable length blocks only

The use of dynamically variable length blocks is significantly more flexible than the two previous approaches, because the block size can be matched to the physical or logical requirements of the devices involved in the transfer. This capability makes more efficient use of bus bandwidth and device time when transferring blocks. On the other hand, the overhead involved in initiating a block transfer would also be expended for single word transfers (blocks of length one). Thus, a compromise between bandwidth and flexibility may have to be arranged, based on the throughput requirements and expected average block size. An example of such a compromise would be a system in which the sizes of the data blocks depended on the source devices. This avoids explicit block length specification, reducing the overhead and improving throughput.

The facility for one-word blocks requires that the control scheme be able to reallocate the bus rapidly enough to minimize wasted bandwidth. Data error response may also be required at the word rate.

Single words or fixed length blocks

In a system where there are high priority devices with low data requirements, this might be a reasonable alternative. The single word option reduces the number of cases where the over-size block would waste bandwidth, buffer space, and device availability, but it still suffers from poor device and bus utilization efficiency when more than one word but less than a block is needed.

The expected mix of block and single word transfers would be a primary influence on the selection of control and communication mechanisms to achieve a proper balance of cost and performance.

Single words or variable length blocks

As might be expected, the capability for both single words and variable length blocks is the most flexible, efficient, and expensive data transfer philosophy. Single words can be handled without the overhead involved in initializing a block transfer. Data blocks can be sized to suit the devices and applications, which optimizes bus usage. The necessity for reassigning the bus as often as every word time imposes a speed constraint on the control method which must be evaluated in light of the expected bus traffic statistics. If data validity response is desired below a message level, the choice of a communication scheme will be affected.

Bus width

The width of a bus impacts many aspects of the system, including cost, reliability, and throughput. Basically, the objective is to achieve the smallest number of lines consistent with the necessary types and rates of communication.

Bus lines require drivers, receivers, cable, connectors, and power, all of which tend to be costly compared to logic. Connectors occupy a significant amount of physical space, and are also among the least reliable components in the system. Reliability is often diminished even further as the number of lines increases due to the additional signal switching noise.

Line combination, serial/parallel conversions, and multilevel encoding are some of the fundamental techniques for reducing bus width. Combination is a method of reducing the number of lines based on function and direction of transmission. Complementary pairs of simplex lines might be replaced with single half-duplex lines. Instead of dedicating individual lines to separate functions, a smaller number of multiplexed lines might be more cost effective, even if extra logic is involved. This includes the performance of bus control functions with coded words on the data lines.

Serial/parallel tradeoffs are frequently employed to balance bus width against system cost and performance. Transmitting fewer bits at a time saves lines, connectors, drivers, and receivers, but adds conversion logic at each end. It may also be necessary to use higher speed (and thus more expensive) circuits to maintain effective throughput. The serial/parallel converters at each end of the bus can be augmented with buffers which absorb traffic fluctuations and allow a lower bandwidth bus. (Independent of bus width considerations, this concept can minimize communication delays due to busy destination devices.) Bit-serial transmission generally is the slowest, requires the most buffering and the least line hardware, produces the smallest amount of noise, and is the most applicable approach in cases with long lines. Parallel transmission is faster, uses more line hardware, generates greater noise, and is more cost-effective over shorter distances.

Multilevel encoding is an approach which converts digital data into analog signals on the bus. It is occasionally used to increase bandwidth by sending parallel data over a single line, but there are numerous disadvantages such as complexity, line voltage drops, lack of noise immunity, etc.

THE SYSTEMATIC APPROACH

A systematic approach to the design of digital bussing structures is outlined in Figure 26. It assumes that pre-

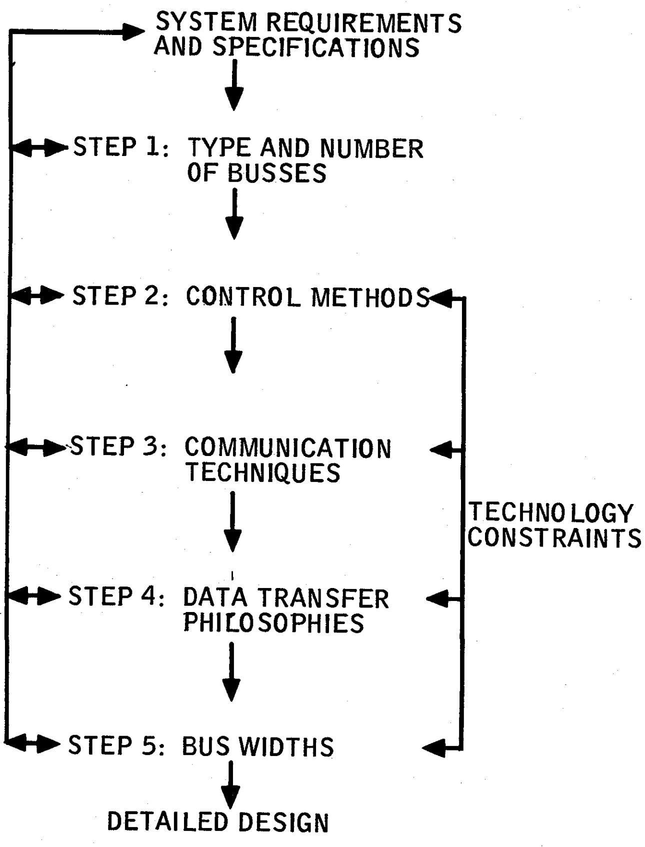

SYSTEM REQUIREMENTS AND SPECIFICATIONS
STEP 1: TYPE AND NUMBER OF BUSSES
STEP 2: CONTROL METHODS
STEP 3: COMMUNICATION TECHNIQUES
STEP 4: DATA TRANSFER PHILOSOPHIES
STEP 5: BUS WIDTHS
DETAILED DESIGN
TECHNOLOGY CONSTRAINTS

Figure 26—Outline of the systematic approach

liminary functional requirements and specifications have been established for the system. The tradeoffs for each bus parameter are interactive, so several iterations are generally necessary. Even the system requirements and specifications may be altered by this feedback in order to achieve an acceptable bus configuration within the technology constraints.

Step 1: Type and number of busses

This is the first and most fundamental step, and involves the specification of dedicated and/or non-dedicated busses. The factors to be considered are: throughput; cost of cables, connectors, etc.; control complexity; communication complexity; reliability; modularity; and bus contention (i.e., availability).

Step 2: Bus control methods

The central choice is among three centralized and three decentralized methods. The Step 1 decision regarding dedicated and non-dedicated busses has a major influence here. The other considerations are: allocation speed; cost of cables, connectors, etc.; control complexity (cost); reliability; modularity; bus contention; allocation flexibility; and device physical placement restrictions.

Step 3: Communication techniques

Either synchronous, asynchronous, or semisynchronous communication techniques may be used, depending on: throughput; cost; reliability; mixed device speeds; bus utilization efficiency; data transfer philosophy; and bus length.

Step 4: Data transfer philosophies

This step is strongly influenced by the need for any block-oriented devices on the bus. In addition, the data transfer philosophy is a function of: control speed; allocation flexibility; control cost; throughput; communication speed; communication technique; device utilization efficiency; and (perhaps) programming and memory allocation.

Step 5: Bus width

Bus width is almost always primarily dictated by either bus length or throughput. Other aspects of this problem are: cost, reliability; communication technique; and communication speed.

CONCLUSION

Historically, many digital bus structures have simply "occurred" ad hoc without adequate consideration of the design tradeoffs and their architectural impacts. This is no longer a viable approach, because systems are becoming more complex and consequently less tolerant of busses which are designed by habit or added as an afterthought. The progress in this area has been hindered by a lack of published literature detailing all the bus parameters and design alternatives. Some aspects of bussing have been touched on briefly as a subsidiary topic in computer architecture papers, and a few concepts have been treated at great length in the substantially different context of communications. In contrast with these foregoing efforts, the intent of this paper is to serve as a step towards a more systematic approach to the entire digital bus structure problem per se.

ANNOTATED BIBLIOGRAPHY

Although many digital designers recognize the importance of bus structures, there have been no previous papers devoted solely to this subject. When bus structures have been discussed in the literature, it has been as a topic subsidiary to other aspects of computer architecture. This section attempts to collect a comprehensive but not exhaustive selection of important papers which deal with various considerations of bus structure design. A guide to the bibliography is given below so that particular facets of this material can be explored. Additionally, each entry has been briefly annotated to provide information on its bus-related contents. The bibliography is grouped into nine categories: Computer Architecture/System Organization, I/O, Sorting Networks, Multiprocessors, Type and Number of Busses, Control Methods, Communication Techniques, Data Transfer Philosophies, and Bus Width.

Computer architecture/system organization

(A2, B2, B4, D2, D3, D4, D7, D8, D9, H3, L1, L4, L7, M4, M5, R5, S7, T4, W1, W5)

Papers in this category basically deal with the architecture of computers and systems, and with how subsystems relate to each other. Alternative architectures (D2, L4, W1) and specific architectures (B2, B4, D3, D4, D7, D8, D9, W5) are discussed. Item A2 is tutorial. The impacts of bus structures (D2, H3, L1) and LSI (L7, M5, R5) on systems organization are described. S7 pursues the effects of new technology on bus structures per se. Report T4 (on which this paper is based) examines the entire bussing problem, and contains a detailed bus design for a specific system.

I/O

(A2, C1, K4)

Several papers deal with bus structures as a subcase of I/O system design. K4 is a tutorial on I/O architecture with many implications on bus structure communication and control. A2 discusses the relationships among the executive, the data bus, and the remainder of the system. C1 considers the overall architecture of an I/O system and its control.

Sorting networks

(B1, L6, T2, T3)

These papers deal with sorting or permuting bus structures. B1 and L6 utilize very simple cells and basically construct their systems from bitonic sorters. T2 utilizes a different approach which is oriented toward ease of implementation with shift registers. T3 employs group theory and a cellular array approach to derive a unique network configuration.

Multiprocessors

(A1, C2, C8, C9, D1, G5)

These papers deal with the design of multiprocessor computer systems. C9 covers the bus architecture of multiprocessors through 1963. A1 describes a multiprocessor with dual non-dedicated busses controlled by a decentralized daisy chain. C2 discusses the relationship between channel rates and memory requirements. C8 and D1 are about multiprocessors using data exchanges. G5 describes a multiprocessor bus that uses associative addressing techniques in its communication portion.

Type and number of busses

(A1, A3, B2, B6, D6, D10, F2, G1, I1, K1, K3, L2, L3, L9, M3, S5, W8, Z2)

The papers in this group describe a computer architecture and include some comments relating to the type and number of busses. Z2 is an example of a dedicated bus, while A1 presents a non-dedicated bus. A1, D10, L2, L3, and Z2 are cases of bus structures with different numbers of busses. B2 points out the hierarchical nature of bus structures. F2 is an example of a store and forward bus structure with dedicated busses and extensive routing control.

Control methods

(A1, A2, B7, P1, P2, P3, Q1, S2, S6, S8, W2, W4, Y1)

The majority of the control techniques are some form of either centralized independent requests (A2, or decentralized daisy chaining (A1). P1 uses polling, and P2 deals with priority control of a system.

Communication techniques

(C6, C7, D5, F1, G3, G4, H2, H4, M1, R2, R3, R4, S1, S3, S4, S9, T1, W6, W7)

These papers tend to be concerned with communication techniques directly rather than as a subsidiary topic. R2 discusses the information lines necessary to communicate in a system. C6, C7, and M1 cover synchronous systems. H4 and S3 are good presentations of the synchronous clock skew problem. S4 deals with the design of a frame and slotted system. F1 describes the use of phase-locked loops for synchronism, while W7 uses bit stuffing for synchronization. The synchronous system in H2 uses a combination of global and local timing. R3 deals with a synchronous system with non-dedicated time slots. D5 contains a good summary of asynchronous communication, and G3 furnishes further examples. G4 points out the importance of communication in digital systems.

Data transfer philosophies

(A4, C1, C3, C4, C5, G2, H1, L5, L8, M2, W3)

Papers in this category are concerned with the philosophies of data transfers. A4 is about transmission error checking and serial-by-byte transmission. C3, C4, and C5 cover buffering and block size from a statistical point of view in simple bus structures such as "loops." G2 studies the choice of block sizes. L5 considers the buffering problem.

Bus width

(B3, B5, G3, C4, C5, K2, R1, T5, Z1)

These papers address the problem of reducing the number of lines in the bus. B3 deals with line drivers and receivers, and contains an extensive bibliography on transmission line papers. B5 discusses balancing the overall system configuration. C3, C4, and C5 are interested in the relationships of burst lengths, number of lines, etc. K2 describes a transmission system utilizing multilevel encoding. T5 is a comprehensive study of line reduction, and includes all the tradeoffs on buffering, multilevel codes, etc., in the design of an actual bus. A machine with a single 200 line bus structure is the topic of R1.

REFERENCES

A1 R L ALONSO et al
A multiprocessing structure
Proceedings IEEE Computer Conference September 1967 pp 56-59
This paper describes a multiprocessor system with non-dedicated instruction and data busses. The control method is a simple decentralized daisy chain.

A2 S J ANDELMAN
Real-time I/O techniques to reduce system costs
Computer Design May 1966 pp 48-54
This article describes two real-time I/O applications and how a computer is used in each. It also indicates the relationships among the system executive, the CPU computations, and the I/O data bus. It includes centralized bus control.

A3 J P ANDERSON et al
D825—a multiple-computer system for command and control
Proceedings FJCC 1962 AFIPS Press pp 86-96
This paper functionally describes the switch interlock system of the Burroughs D825 system. The switch is essentially a crossbar which can handle up to 64 devices. A priority-oriented bus allocation mechanism handles conflicting allocation requests. Priorities are preemptive.

A4 A AVIZIENIS
Design of fault-tolerant computers
Proceedings FJCC 1967 AFIPS Press pp 733-743
This paper describes the internal structure of the JPL-STAR computer. The bus structure consists of two busses and two bus checkers. The busses transmit information in four-bit bytes and the bus checkers check for transmission errors.

B1 K E BATCHER
Sorting networks and their application
Proceedings SJCC 1968 AFIPS Press pp 307-314
This paper describes various configurations of bitonic sorting networks which can be utilized as routing networks or permutation switches in multiprocessor systems.

B2 H R BEELITZ
System architecture for large-scale integration
Proceedings FJCC 1967 AFIPS Press pp 185-200
This paper describes the architecture of LIMAC. It notes the hierarchical nature of bus structures, stating, "A *local* bus structure interconnects the sub-partitions of a functional module in the same sense that the machine bus interconnects all functional modules."

B3 R O BERG et al
PEPE implementation study
Honeywell Report 12251-FR Prepared for System Development Corporation under Subcontract SDC 71-61
This report contains an extensive bibliography of signal transmission papers and a survey of line drivers and receivers. It also describes the bus designs for the PEPE multiprocessor system.

B4 N A BOEHMER et al
Advanced avionic digital computer—arithmetic and control unit design
Hughes Aircraft Report P70-517 prepared under Navy contract N62269-70-C-0534 December 1970
This report describes a main data bus design for the Advanced Avionic Digital Computer, including the bus communication and allocation mechanisms.

B5 F P BROOKS K E IVERSON
Automatic data processing
Wiley New York 1969 Section 5.4 Parameters of computer organization pp 250-262
This section descusses speed/cost/balance tradeoffs in computer architecture. Of specific interest is how bus width, speed, and degree of parallelism affect computer performance. Examples of tradeoff results are given in terms of the System/360.

B6 W BUCHHOLZ
Planning a computer system
McGraw-Hill New York 1962
Chapter 16 of this book describes the data exchange of the STRETCH computer. The data exchange is a switched bus which handles data flow among I/O and external storage units and the primary store. It is independent of CPU processes and able to function concurrently with the central processor.

B7 H B BURNER et al
A programmable data concentrator for a large computing system
IEEE Transactions on Computers November 1969 pp 1030-1038
This paper describes the internal structure of a data concentrator to be used with an IBM 360/67. The concentrator utilizes an Interdata Model 4 computer. The details of the bus structure, including timing and control signals, are given. The system was built and utilized at Washington State University, Pullman, Washington.

C1 G N CEDARQUIST
An input/output system for a multiprogrammed computer
Report No 223 April 1967 Department of Computer Science University of Illinois
This report describes the architecture of I/O systems, and deals with some parameters of bus structures through discussion of data transfers. It is primarily concerned with the implementation of centralized control and communication logic.

C2 Y C E CHEN D L EPLEY
Bounds on memory requirements of multiprocessing systems
Proceedings 6th Annual Allerton Conference on Circuit and System Theory October 1968 pp 523-531
This paper presents a model of a multiprocessor with a multilevel memory. Given a computation graph with specified execution times and main memory requirements, bounds on the required main memory and the inter-memory channel rates are calculated. The trade-off between main memory size and backing memory channel capacity is discussed at some length.

C3 W W CHU
A study of asynchronous time division multiplexing for time sharing computer systems
Proceedings FJCC 1969 AFIPS Press pp 669-678

This paper describes the use of an asynchronous time division multiplexing system. A model is given which relates buffer size and queuing delays to traffic, number of lines, and burst lengths.

C4 W W CHU
Demultiplexing considerations for statistical multiplexers
IEEE Transactions on Computers June 1972 pp 603-609
This paper discusses tradeoffs and simulation results useful in the design of buffers used in a computer communication system. The tradeoffs between message lengths, buffer size, traffic intensity, etc., are considered.

C5 W W CHU A G KONHEIM
On the analysis and modeling of a class of computer communication systems
IEEE Transactions on Communications June 1972 pp 645-660
This paper derives models for a computer communication environment, applied to star and loop bus structure systems. The model provides a means of relating statistical parameters for traffic intensities, message lengths, etc.

C6 N CLARK A C GANNET
Computer-to-computer communication at 2.5 megabit/sec
Proceedings of IFIP Congress 62 North Holland Publishing Company September 1962 pp 347-353
This paper describes an experimental synchronous high speed (2.5 megabit/second) communication system. It indicates the relationships of all system parts necessary to communicate in a party-line fashion among three computers.

C7 COLLINS RADIO CORPORATION
C-system overview 523-0561644-001736 Dallas Texas October 1 1969
This brochure describes the architecture of the Collins C-System, especially the design and features of the Time Division Exchange (TDX) loop. The TDX loop is a 32 million bit-per-second serial communication link. Communication between devices is at a 2 million word-per-second rate. The system as initially implemented contained 16 channels, with expansion to a 512 million bit-per-second capability envisioned.

C8 M E CONWAY
A multiprocessor system design
Proceedings FJCC 1963 AFIPS Press pp 139-146
This paper describes the design of a multiprocessor system which useds a matrix switch (called a memory exchange) to connect processors to memories. The unique feature of the configuration is that an associative memory is placed between each processor and the memory exchange for addressing purposes.

C9 A J CRITCHLOW
Generalized multiprocessing and multiprogramming systems
Proceedings FJCC 1963 AFIPS Press pp 107-126
This paper describes the state of development of multiprocessor systems in 1963. There were essentially three bus schemes in use: the crossbar switch (Burroughs D825), the multiple bus (CDC-3600) and the time-shared bus (IBM STRETCH). Functional descriptions of the bus concepts are presented.

D1 R L DAVIS et al
A building block approach to multiprocessing
Proceedings FJCC 1972 AFIPS Press pp 685-703
This paper describes a bus structure (called a Switch Interlock) for use in a multiprocessor. It discusses the tradeoffs in choosing the structure, and looks at single bus, multiple bus, multiport, and crossbar systems. The Switch Interlock is a dedicated bus matrix switch which supports both single word and block transfers. The switch is designed to be implemented for bus widths from bit-serial to fully word-parallel.

D2 A J DEERFIELD
Architectural study of a distributed fetch computer
NAECON 1971 Record pp 214-217
This paper describes the distributed fetch computer in which the fetch (procedure and data) portion of the machine is distributed to the memory modules.

D3 A J DEERFIELD et al
Distributed fetch computer concept study
Air Force Contract No F-71-C-1417 February 1972
This report describes the design of a bus structure for use in the distributed fetch computer. This machine repartitions the fetch and execute portions of the processor in a multiprocessor system. The fetch units are associated with the memories instead of being with the execute units, thus decreasing bus traffic.

D4 A J DEERFIELD et al
Interim report for arithmetic and control logic design study
Navy Contract N62269-72-C-0023 May 1972
This report describes a proposed bus structure for the Advanced Avionic Digital Computer and some of the tradeoffs considered during the design.

D5 J B DENNIS S S PATIL
Computation structures
Chapter 4—Asynchronous Modular Systems
MIT Department of Electrical Engineering Cambridge Massachusetts
This chapter describes the reasons for asynchronous systems, and gives examples of asynchronous techniques and their timing mechanisms. It is useful in understanding asynchronous communications.

D6 E W DEVORE D H LANDER
Switching in a computer complex for I/O flexibility
1964 NEC pp 445-447
This paper describes the IBM 2816 Switching Unit, the bus system utilized to interconnect CPU's and tape drives. It discusses the modularity tradeoffs made in the 2816.

D7 DIGITAL EQUIPMENT CORPORATION
PDP-11 handbook
Chapter 8—Description of the UNIBUS pp 59-68 Maynard Massachusetts 1969
This chapter of the PDP-11 user's manual describes the UNIBUS functionally as a subsystem of the PDP-11. Data transfer operations performed by the bus are described and illustrated with examples, along with general concepts of bus operation and control.

D8 DIGITAL EQUIPMENT CORPORATION
PDP-11 interface
Application Note Maynard Massachusetts
This document gives a brief description of the PDP-11 UNIBUS, a single undedicated bus with centralized daisy-chain control and fully-interlocked request/acknowledge communication.

D9 DIGITAL EQUIPMENT CORPORATION
PDP-11 unibus interface manual
DEC-11-HIAB-D Maynard Massachusetts 1970
This manual gives a detailed description of the PDP-11 UNIBUS, its operation in the computer, and methods for interfacing peripheral equipment to the bus.

D10 S B DINMAN
Direct function processor concept for system control
Computer Design March 1970 pp 55-60
This article describes the (patented) GRI-909 bus structure.

The machine consists of a series of functional modules strung between two undedicated busses with a bus modifier unit (which serves a function similar to the alpha code on the Harvard MARK IV). The GRI-909 is quite similar to the DEC PDP-11.

F1 K FERTIG B C DUNCAN
A new high-speed general purpose input/output mechanism with real-time computing capability
Proceedings FJCC 1967 AFIPS Press pp 281-289
This paper describes techniques for I/O processing of self-clocked data utilizing phase locked loops.

F2 H FRANK et al
Computer communication network design—experience with theory and practice
SJCC 1972 AFIPS Press pp 255-270
This paper describes the ARPANET design from the vantage point of two years experience with the message switching system. ARPANET is a store and forward message switching network in which a device interfaces into the system by means of an interface message processor (IMP). The IMP then routes the message through the network topology. This paper provides insight into the design and specification of dedicated "store-and-forward" message switching systems.

G1 E C GANGL
Modular avionic computer
NAECON 1972 Record pp 248-251
This paper describes the architecture of a modular computer including its internal bus structure. The bus consists of four parallel segments: a data bus, a status bus, a microprogrammed command bus, and a power distribution bus.

G2 D H GIBSON
Considerations in block oriented systems design
Proceedings SJCC 1967 AFIPS Press pp 75-80
This paper describes the rationale and techniques for block transfers between CPU and memory. The study is to determine the affect of block size on CPU throughput.

G3 A I GROUDAN
The SKC-2000 advanced aerospace computer
NAECON 1972 Record pp 229-235
This paper describes the SKC-2000 computer and its internal bus structure. The bus operates in a request/acknowledge mode of communication and can handle devices of different speeds from 1 microsecond to larger than a millisecond with no design changes.

G4 H W GSCHWIND
Design of digital computers
Communications in Digital Computer Systems Chapter 8 Section 5 Springer-Verlag New York 1967 pp 347-367
This section describes computer I/O and access paths (busses) in terms of their communication ramifications. It points out that "even experts failed to look at computers seriously from a communication point of view for a surprisingly long time." It also details the communication that occurs in some general computer configurations.

G5 D C GUNDERSON
Multi-processor computing apparatus
U S Patent 3521238 July 13 1967
This patent describes a method of bussing in a multiprocessor system based upon the use of an associative switch. This bus scheme allows processors to access a centralized system memory by either location or some property of the data (content addressability). Each processor has its own individual access to the system memory so the bus is very reliable.

H1 M L HANSON
Input/output techniques for computer communication
Computer Design June 1969 pp 42-47
This article describes the I/O systems in several UNIVAC machines, and considers the types of data transfers, staus words, number of lines, method of operation, etc., of these bus structures.

H2 R H HARDIN
Self sequencing data bus technique for space shuttle
Proceedings Space Shuttle Integrated Electronic Conference Vol 2 1971 pp 111-139
This presentation describes the design of SLAT (Slot Assigned TDM), a data bus for space shuttle. SLAT is a synchronous bus with global plus local synchronization. The requirements, length, control method, clock skew, and synchronization tradeoffs are discussed.

H3 H HELLERMAN
Digital computer system principles
Data Flow Circuits and Magnetic-Core Storage
McGraw-Hill New York 1967 Chapter 5 pp 207-235
This chapter contains a discussion of data flow or bus circuits, with special emphasis on the trade-offs possible between economy and speed. The author stresses the fact that the bus organization of a computer is a major factor determining its performance.

H4 G P HYATT
Digital data transmission
Computer Design Vol 6 No 11 November 1967 pp 26-30
This article deals primarily with the transmission of data in a synchronous bus structure. It considers in detail the clock skew problem, and describes propagation delay and mechanization problems. It concludes that the clock pulse should not be daisy-chained, but radially distributed, and that the sum (worst case) of data propagation delays must be less than the clock pulse period.

I1 F INOSE et al
A data highway system
Instrumentation Technology January 1971 pp 63-67
This article describes a data bus designed to interface many digital devices together. The system is essentially a nondedicated single bus with one wire for data and another for addresses. The system is connected together in a "loop configuration." It uses a "5-value pulse" for synchronization, etc. The system has an access time of 200 microseconds and can handle 100 devices on a bus up to 1 kilometer in length.

K1 J C KAISER J GIBBON
A simplified method of transmitting and controlling digital data
Computer Design May 1970 pp 87-91
This article treats the tradeoffs between the number of parallel lines in a bus and the complexity of gating at the bus destinations. The authors develop a matrix switch concept as a data exchange under program control. The programmed instruction thus is able to dynamically interconnect system elements by coded pulse coincidence control of the switching matrix.

K2 H KANEKO A SAWAI
Multilevel PCM transmission over a cable using feedback balanced codes
NEC 1967 pp 508-513
This paper describes a multilevel PCM code (Feedback Balanced Code) suitable for transmission of data on a coaxial transmission cable.

K3 L J KOCZELA
Distributed processor organization
Advances in Computers Vol 19 Chapter 7 Communication Busses Academic Press New York 1968 pp 346-349
This author presents a functional description of a bussing scheme for a distributed cellular computer. Each processor can address its own private memory plus bulk storage. Communication between cells takes place over the bus in two modes: Local (between two cells) and Global (controller call plus one or more controlled cells). The intercell bus is used for both instructions and data; all transfers are set up and directed by the controller cell by means of eight bus control commands.

K4 G A KORN
Digital computer interface systems
Simulation December 1968 pp 285-298
This paper is a tutorial on digital computer interfaces. It begins with the party line I/O bus, and covers how devices are controlled, how interrupts are handled, and how data channels operate. It discusses the overall subject of interfaces (I/O and bussing system) from the systems point of view, describing how the subsystems all relate to each other.

L1 J R LAND
Data bus concepts for the space shuttle
Proceedings Space Shuttle Integrated Electronic Conference Vol 3 1971 pp 710-785
This presents the space shuttle data management computer architecture from a bus-oriented viewpoint. It discusses the properties and design characteristics of the bus structures, and summarizes the design and mechanization trade-offs.

L2 F J LANGLEY
A universal function unit for avionic and missile systems
NAECON Record 1971 pp 178-185
This paper discusses some trade-offs in computer architectures, and categorizes some architectures by their bus structures, providing an example for each category. It considers single time-shared bus systems, multiple bus systems, crossbar systems, dual bus external ensemble systems, multiple-bus integrated ensemble systems, etc.

L3 R LARKIN
A mini-computer multiprocessing system
Second Annual Computer Designers Conference Los Angeles California February 1971 pp 231-235
The topology of communication between computer subsystems is discussed. Six basic topologies for communication internal to a computer are described: (1) radial, (2) tree, (3) bus, (4) matrix, (5) iterative, and (6) symmetric. Some topological implications of bus structures are discussed including the need to insure positive (one device) control of the bus during its transmission phase. All six topologies can be expressed in terms of dedicated and non-dedicated bus structures.

L4 S E LASS
A fourth generation computer organization
Proceedings SJCC 1968 AFIPS Press pp 435-441
This paper functionally describes the internal organization of a "fourth-generation" computer including its data channels and I/O bus structure.

L5 A L LEINER
Buffering between input/output and the computer
Proceedings FJCC 1962 pp 22-31
This paper describes the tradeoffs in synchronizing devices, and considers solutions to the problem of buffering between devices of different speeds.

L6 K N LEVITT
A study of data communication problems in a self-repairable multiprocessor
Proceedings SJCC 1968 AFIPS Press pp 515-527
This paper presents a method of aerospace multiprocessor reliability enhancement by dynamic reconfiguration using busses which are data commutators. Two realizations of such a bus technique are permutation switching networks and crossbar switches.

L7 S Y LEVY
Systems utilization of large-scale integration
IEEE Transactions on Computers Vol EC-16 No 5 1967 pp 562-566
This paper describes a new approach to computer organization based on LSI technology, employing functional partitioning of both the data path and control. Of particular interest is the data bus structure of an RCA Laboratories experimental machine using LSI technology.

L8 W A LEVY E W VEITCH
Design for computer communication systems
Computer Design January 1966 pp 36-41
This article relates memory size considerations to a user's wait time for a line to the memory. It is applicable to bus bandwidth design in the analysis of buffer sizes needed to load up a bus structure.

L9 R C LUTZ
PCM using high speed memory system for switching applications
Data and Communication Design May-June 1972 pp 26-28
This article details a method of replacing a crossbar switch with a memory having an input and output commutation system and some counting logic. Advantages of this approach are low cost and linear growth.

M1 J S MAYO
An approach to digital system network
IEEE Transactions on Communication Technology April 1967 pp 307-310
This paper deals with synchronizing communication between devices with unlocked clocks. A system with frame sync is postulated and the number of bits necessary for efficient pulse stuffing is derived.

M2 J D MENG
A serial input/output scheme for small computers
Computer Design March 1970 pp 71-75
This article describes the trade-offs and results of designing an I/O data bus structure for a minicomputer.

M3 J S MILLER et al
Multiprocessor computer system study
NASA Contract No 9-9763 March 1970
This report reviews the number and type of busses used in several computing systems such as: CDC 6000, IBM DCS, IBM 360 ASP series, IBM 4-Pi, Burroughs D825 and 5500, etc. It goes on to suggest the design of a multiprocessor for a space station. In particular the system has two busses, one for I/O and one for internal transfers. Specifically described are: message structure, access control, error checking and required bandwidth. A 220 MHz bandwidth requirement is deduced.

M4 W F MILLER R ASCHENBRENNER
The GUS multicomputer system
IEEE Transactions on Computers December 1963 pp 671-676
This paper describes an Argonne Lab experimental computer with several memory and processing subsystems. All internal memory communication is handled by the Dis-

tributor, which functions as a data exchange and is expandable. No detailed description of the Distributor operation is furnished.

M5 R C MINNICK et al
Cellular bulk transfer systems
Air Force Contract No F19628-67-C-0293 3 AD683744 October 1968
Part C of this report describes a bulk transfer system composed of an input array, an output array, and a mapping device. The mapping device moves data from the input to the output array and may contain logic. Simple bulk transfer systems are described which perform permutation on the data during its mapping.

P1 P E PAYNE
A method of data transmission requiring maximum turnaround time
Computer Design November 1968 p 82
This article describes a method of controlling data transmission between devices by polling.

P2 M PIRTLE
Intercommunication of processors and memory
Proceedings FJCC 1967 AFIPS Press pp 621-633
This paper discusses the throughput of several different bus structures in a system configuration with the intent of providing the appropriate amount of memory bandwidth. It describes the allocation sequence of a typical bus, and concludes that it can be very effective to assign "... priorities to requests, rather than to processors and busses ... with memory systems which provide ample memory bus bandwidth to the processors."

P3 W W PLUMMER
Asynchronous arbiters
Computation Structures Group Memo No 56 MIT Project MAC February 1971
This memo describes logic for determining which of several requesting CPU's get access and in what order to a memory. It is potentially a portion of the control logic for a bus structure, and describes several different algorithms for granting access.

Q1 J T QUATSE et al
The external access network of a modular computer system
Proceedings SJCC 1972 AFIPS Press pp 783-790
This paper describes the External Access Network (EAN), a switching network designed to interface processors to processors, processors to facilities, and memory to facilities in a modular time sharing system (PRIME) being built at Berkeley. The EAN acts like a crossbar switch or data exchange, and consists of processor, device, and switch nodes. To communicate, a processor selects an available switch node and connects the appropriate device node to it.

R1 R RICE W R SMITH
SYMBOL—a major departure from classic software dominated Von Neumann computing systems
Proceedings SJCC 1971 AFIPS Press pp 575-587
This paper describes a functionally designed bus-oriented system. The system bus consists of 200 interconnection lines which run the length of the mainframe.

R2 R RINDER
The input/output architecture of minicomputers
Datamation May 1970 pp 119-124
This article surveys the architecture of minicomputer I/O units. It describes a typical I/O bus and the lines of information it would carry.

R3 M P RISTENBATT D R ROTHSCHILD
Asynchronous time multiplexing
IEEE Transactions on Communication Technology June 1968 pp 349-357
This paper describes the use of "asynchronous time multiplexing" techniques on analog data. Basically, the paper describes a synchronous system with non-dedicated time slots.

R4 K ROEDL R STONER
Unique synchronizing technique increases digital transmission rate
Electronics March 15 1963 pp 75-76
This note provides a method of synchronizing two devices having local clocks of supposedly equal frequencies.

R5 K K ROY
Cellular bulk transfer system
PhD Thesis Montana State University Bozeman Montana March 1970
Bulk transfer systems composed of input logic, output logic, and a mapping device are studied. The influences of mapping device, parallelism, etc., are considered.

S1 T SAITO H INOSE
Computer simulation of generalized mutually synchronized systems
Symposium on Computer Processing in Communications Polytechnic Institute of Brooklyn April 1969 pp 559-577
This paper describes ten ways to mutually synchronize devices having separate clocks so that data can be accurately delivered in the correct time slot of a synchronous system. The results of the simulation relate to the stability of the synchronizing methods.

S2 J SANTOS M I OTERO
On transferences and priorities in computer networks
Symposium on Computers and Automata Vol 21 1971 pp 265-275
The structure of bus (channel) controllers is considered using the language of automata theory. The controller is decomposed into two units: one receives requests and availability signals, and generates corresponding requests to the other unit which allocates the bus on a priority basis. Both units are further decomposed into subunits.

S3 J W SCHWARTZ
Synchronization in communication satellite systems
NEC 1967 pp 526-527
This paper describes tradeoffs and potential solutions to the clock skew problem in a widely dispersed system.

S4 C D SMITH
Optimization of design parameters for serial TDM
Computer Design January 1972 pp 51-54
This article derives analytical tools for the analysis and optimization of a synchronous system with global plus local timing.

S5 D J SPENCER
Data bus design techniques
NASA TM-X-52876 Vol VI pp 95-113
This paper discusses design alternatives for a multiplexed data bus to reduce point-to-point wiring cost and complexity. The author investigates coupling, coding, and control factors for both low and high signal-to-noise ratio lines for handling a data rate less than five million bits per second.

S6 D C STANGA
Univac 1108 multiprocessor system
Proceedings SJCC 1971 AFIPS Press pp 67-74
This paper describes how memory accesses are made from the multiple processors to the multiple memory banks in the 1108 multiprocessor system. It gives a block diagram of the

system interconnectivity and describes how the multiple module access units operate to provide multiple access paths to a memory module.

S7 D J STIGLIANI et al
Wavelength division multiplexing in light interface technology
AD-721085 March 1971
This report describes the fabrication of a five-channel optical multiplexed communication line, and suggests some alternatives for matching wavelength multiplexed light transmission times to digital electrical circuits.

S8 J N STURMAN
An iteratively structured general purpose digital computer
IEEE Transactions on Computers January 1968 pp 2-9
This paper describes a bus and its use in an iterative computer. The system is a dual dedicated bus structure with centralized control.

S9 J N STURMAN
Asynchronous operation of an iteratively structured general purpose digital computer
IEEE Transactions on Computers January 1968 pp 10-17
This paper describes the synchronization of an iterative structure computer. The processing elements are connected on a common complex symbol bus. To allow asynchronous operation, a set of timing busses are added to the system common complex symbol bus. The timing busses take advantage of their transmission line properties to provide synchronism of the processors.

T1 F W THOBURN
A transmission control unit for high speed computer-to-computer communication
IBM Journal of Research and Development November 1970 pp 614-619
This paper describes a multiplex bus system for connecting a large number of computers together in a star organization. Special emphasis is given to the transmission control unit, a microprogrammed polling and interface unit which uses synchronous two-frequency modulation and a serializer/de-serializer unit.

T2 K J THURBER
Programmable indexing networks
Proceedings SJCC 1970 AFIPS Press pp 51-58
This paper describes data routing networks designed to perform a generalized index on the data during the routing process. The indexing networks map an input vector onto an output vector. The mapping is arbitrary and programmable. Several different solutions are presented with varying hardware, speed, and timing requirements. The networks are described in terms of shift register implementations.

T3 K J THURBER
Permutation switching networks
Proceedings of the 1971 Computer Designer's Conference Industrial and Scientific Conference Management Chicago Illinois January 1971 pp 7-24
This paper describes several permutation networks designed to provide a programmable system capable of interconnecting system elements. The networks are partitioned for LSI implementation and can be utilized in a pipeline fashion. Algorithms are given to determine a program to produce any of the N! possible permutations of N input lines.

T4 K J THURBER et al
Master executive control for AADC
Navy Contract N62269-72-C-0051 June 18 1972
This report describes a systematic approach to the design of digital bus structures and applies this tool to the design of a bus structure for the Advanced Avionic Digital Computer. The structure is designed with three major requirements: flexibility, modularity, and reliability.

T5 A TURCZYN
High speed data transmission scheme
Proceedings 3rd Univac DPD Research and Engineering Symposium May 1968
The increasing complexity of multiprocessor computer systems with a high degree of parallelism within the computer system has created major internal communication problems. If each processing unit should be able to communicate with many other subsystems, the author recommends either a data exchange, or switching center, or parallel point-to-point wiring. The latter has the advantage of fast transfer and minimal data registers, but in a multiprocessor it results in a large number of cables. This paper discusses the state-of-the-art of internal multiplexing and multi-level coding schemes for reducing the number of lines in the system.

W1 E G WAGNER
On connecting modules together uniformly to form a modular computer
IEEE Transactions on Computers December 1966 pp 864-872
This paper provides mathematical group theoretic precision to the idea of uniform bus structure in cellular computers.

W2 P W WARD
A scheme for dynamic priority control in demand actuated multiplexing
IEEE Computer Society Conference Boston September 1971 pp 51-52
This paper describes a priority conflict resolution method which is used in an I/O multiplexer system.

W3 R WATSON
Timesharing system design concepts
Chapter 3—Communications McGraw-Hill 1970 pp 78-110
This chapter provides a summary of "communication" among memories, processors, IOP's, etc. The discussion is oriented toward example configurations. Subjects discussed are: (1) use of multiple memory modules, interleaving, and buffering to increase memory bandwidth; (2) connection of subsystems using direct connections, crossbar switches, multiplexed busses, etc.; and (3) the transmission medium. Items discussed under transmission medium are synchronous and asynchronous transmission, line types (simplex, half-duplex, and full-duplex), modulation, etc.

W4 D R WELLER
A loop communication system for I/O to a small multi-user computer
IEEE Computer Society Conference Boston September 1971 pp 49-50
This paper describes a single-line non-dedicated bus with daisy-chained control for the DDP-516 computer. Message format and speed of operation are detailed.

W5 G P WEST R J KOERNER
Communications within a polymorphic intellectronic system
Proceedings of Western Joint Computer Conference San Francisco May 3-5 1960 pp 225-230
This paper describes a crosspoint data exchange used in the RW-400 computer. The switch was mechanized using transfluxor cores.

W6 L P WEST
Loop-transmission control structures
IEEE Transactions on Communications June 1972 pp 531-539
This paper considers the problem of transmitting data on a

communication loop. It discusses time slots, frame pulses, addressing techniques, and efficiency of utilization. It also discusses a number of ways for assigning time slots for utilization on the impact of slot size on loop utilization efficiency.

W7 M W WILLARD L J HORKAN
Maintaining bit integrity in time division transmission
NAECON 1971 Record pp 240-247
This paper describes the tradeoffs involved in synchronizing high speed digital subsystems which are communicating over large distances. It considers clocking and buffering tradeoffs.

W8 D R WULFINGHOFF
Code activated switching—a solution to multiprocessing problems
Computer Design April 1971 pp 67-71
The author points out that multiprocessor computer configurations have a large number of interconnections between elements causing considerable hardware and software complexity. He describes a technique whereby each program to be run is assigned a code, identifier, or signature; then when this program is activated the system resources it requires can be "lined-up" for use. He compares this scheme to that employed for telephone switching. Code activated switching is illustrated by two system block diagrams: a special purpose control computer and a general purpose time-shared computer.

Y1 B S YOLKEN
Data bus—method for data acquisition and distribution within vehicles
NAECON 1971 Record pp 248-253
This paper discusses a time division multiplexed bus, and considers bus control, bit synchronization, and technology tradeoffs.

Z1 R E ZIMMERMAN
The structure and organization of communication processors
PhD Dissertation Electrical Engineering Department University of Michigan September 1971
This dissertation describes a multi-bus computer used as a terminal processor. It has a pair of instruction busses which start and then signal completion of processes performed in functional units or subsystems. The machine has three data busses: a memory bus which serves as the primary system communication bus, a flag address bus, and a flag data bus. All busses are eight bits wide and the three data busses are bidirectional.

Z2 R J ZINGG
Structure and organization of a pattern processor for hand-printed character recognition
PhD Dissertation Iowa State University Ames Iowa 1968
This dissertation describes a bus-oriented special purpose computer designed for research in character recognition. The machine contains a control bus, a scratchpad memory bus, and three data busses. Each register that can be reached by a data bus has two control flip-flops associated with it and these determine to which data bus it is to be connected. These connections are controlled by a hardware command. The contents of several registers can be placed on one data bus to yield a bit-by-bit logical inclusive OR. Also, the contents of one data bus can be transferred to several registers and the contents of all three busses transferred in parallel under program command. This processor is a rather interesting example of a five bus processor.

Improvements in the design and performance of the ARPA network

by J. M. McQUILLAN, W. R. CROWTHER, B. P. COSELL, D. C. WALDEN, and F. E. HEART

Bolt Beranek and Newman Inc.
Cambridge, Massachusetts

INTRODUCTION

In late 1968 the Advanced Research Projects Agency of the Department of Defense (ARPA) embarked on the implementation of a new type of computer network which would interconnect, via common-carrier circuits, a number of dissimilar computers at widely separated, ARPA-sponsored research centers. The primary purpose of this interconnection was resource sharing, whereby persons and programs at one research center might access data and interactively use programs that exist and run in other computers of the network. The interconnection was to be realized using wideband leased lines and the technique of message switching, wherein a dedicated path is not set up between computers desiring to communicate, but instead the communication takes place through a sequence of messages each of which carries an address. A message generally traverses several network nodes in going from source to destination, and at each node a copy of the message is stored until it is safely received at the following node.

The ARPA Network has been in operation for over three years and has become a national facility. The network has grown to over thirty sites spread across the United States, and is steadily growing; over forty independent computer systems of varying manufacture are interconnected; provision has been made for terminal access to the network from sites which do not enjoy the ownership of an independent computer system; and there is world-wide excitement and interest in this type of network, with a number of derivative networks in their formative stages. A schematic map of the ARPA Network as of the fall of 1972 is shown in Figure 1.

As can be seen from the map, each site in the ARPA Network consists of up to four independent computer systems (called Hosts) and one communications processor called an Interface Message Processor, or IMP. All of the Hosts at a site are directly connected to the IMP. Some IMPs also provide the ability to connect terminals directly to the network; these are called Terminal Interface Message Processors, or TIPs. The IMPs are connected together by wideband telephone lines and provide a subnet through which the Hosts communicate. Each IMP may be connected to as many as five other IMPs using telephone lines with bandwidths from 9.6 to 230.4 kilobits per second. The typical bandwidth is 50 kilobits.

During these three years of network growth, the actual user traffic has been light and network performance under such light loads has been excellent. However, experimental traffic, as well as simulation studies, uncovered logical flaws in the IMP software which degraded performance at heavy loads. The software was therefore substantially modified in the spring of 1972. This paper is largely addressed to describing the new approaches which were taken.

The first section of the paper considers some criteria of good network design and then presents our new algorithms in the areas of source-to-destination sequence and flow control, as well as our new IMP-to-IMP acknowledgment strategy. The second section addresses changes in program structure; the third section re-evaluates the IMP's performance in light of these changes. The final section mentions some broader issues.

The initial design of the ARPA Network and the IMP was described at the 1970 Spring Joint Computer Conference,[1] and the TIP development was described at the 1972 Spring Joint Computer Conference.[2] These papers are important background to a reading of the present paper.

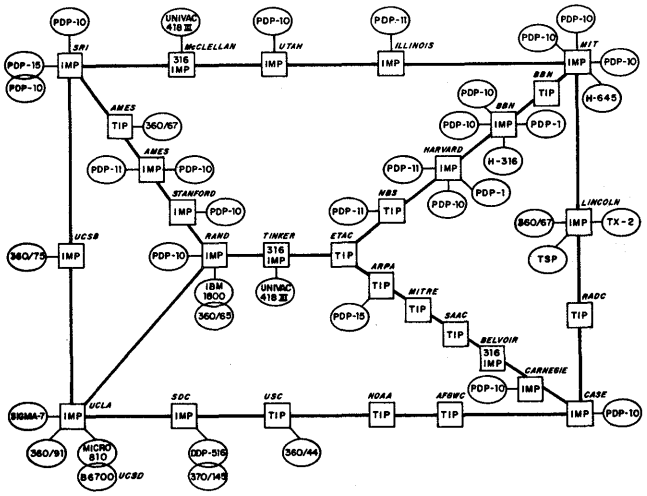


Figure 1—ARPA network, logical map, August 1972

NEW ALGORITHMS

A balanced design for a communication system should provide quick delivery of short interactive messages and high bandwidth for long files of data. The IMP program was designed to perform well under these bimodal traffic conditions. The experience of the first two and one half years of the ARPA Network's operation indicated that the performance goal of low delay had been achieved. The lightly-loaded network delivered short messages over several hops in about one-tenth of a second. Moreover, even under heavy load, the delay was almost always less than one-half second. The network also provided good throughput rates for long messages at light and moderate traffic levels. However, the throughput of the network degraded significantly under heavy loads, so that the goal of high bandwidth had not been completely realized.

We isolated a problem in the initial network design which led to degradation under heavy loads.[3,4] This problem involves messages arriving at a destination IMP at a rate faster than they can be delivered to the destination Host. We call this *reassembly congestion*. Reassembly congestion leads to a condition we call *reassembly lockup* in which the destination IMP is incapable of passing any traffic to its Hosts. Our algorithm to prevent reassembly congestion and the related sequence control algorithm are described in the following subsections.

We also found that the IMP and line bandwidth requirements for handling IMP-to-IMP traffic could be substantially reduced. Improvements in this area translate directly into increases in the maximum throughput rate that an IMP can maintain. Our new algorithm in this area is also given below.

Source-to-destination flow control

For efficiency, it is necessary to provide, somewhere in the network, a certain amount of buffering between the source and destination Hosts, preferably an amount equal to the bandwidth of the channel between the Hosts multiplied by the round trip time over the channel. The problem of flow control is to prevent messages from entering the network for which network buffering is not available and which could congest the network and lead to reassembly lockup, as illustrated in Figure 2.

In Figure 2, IMP 1 is sending multi-packet messages to IMP 3; a lockup can occur when all the reassembly buffers in IMP 3 are devoted to partially reassembled messages A and B. Since IMP 3 has reserved all its remaining space for awaited packets of these partially reassembled messages, it can only take in those particular packets from IMP 2. These outstanding packets, however, are two hops away in IMP 1. They cannot get through because IMP 2 is filled with store-and-forward packets of messages C, D, and E (destined for IMP 3) which IMP 3 cannot yet accept. Thus, IMP 3 will never be able to complete the reassembly of messages A and B.

The original network design based source-to-destination sequence and flow control on the link mechanism previously reported in References 1 and 5. Only a single message on a given link was permitted in the subnetwork at one time, and sequence numbers were used to detect duplicate messages on a given link.

We were always aware that Hosts could defeat our flow control mechanism by "spraying" messages over an inordinately large number of links, but we counted on the nonmalicious behavior of the Hosts to keep the

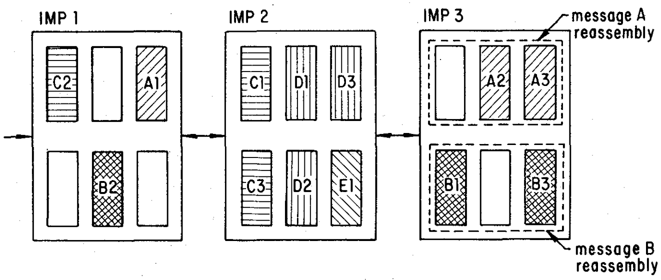


Figure 2—Reassembly lockup

number of links in use below the level at which problems occur. However, simulations and experiments artificially loading the network demonstrated that communication between a pair of Hosts on even a modest number of links could defeat our flow control mechanism; further, it could be defeated by a number of Hosts communicating with a common site even though each Host used only one link. Simulations[3,4] showed that reassembly lockup may eventually occur when over five links to a particular Host are simultaneously in use. With ten or more links in use with multipacket messages, reassembly lockup occurs almost instantly.

If the buffering is provided in the source IMP, one can optimize for low delay transmissions. If the buffering is provided at the destination IMP, one can optimize for high bandwidth transmissions. To be consistent with our view of a balanced communications system, we have developed an approach to reassembly congestion which utilizes some buffer storage at both the source and destination; our solution also utilizes a request mechanism from source IMP to destination IMP.*

Specifically, no multipacket message is allowed to enter the network until storage for the message has been allocated at the destination IMP. As soon as the source IMP takes in the first packet of a multipacket message, it sends a small control message to the destination IMP requesting that reassembly storage be reserved at the destination for this message. It does not take in further packets from the Host until it receives an allocation message in reply. The destination IMP queues the request and sends the allocation message to the source IMP when enough reassembly storage is free; at this point the source IMP sends the message to the destination.

We maximize the effective bandwidth for sequences of long messages by permitting all but the first message to bypass the request mechanism. When the message itself arrives at the destination, and the destination IMP is about to return the Ready-For-Next-Message (RFNM), the destination IMP waits until it has room for an additional multipacket message. It then piggybacks a storage allocation on the RFNM. If the source Host is prompt in answering the RFNM with its next message, an allocation is ready and the message can be transmitted at once. If the source Host delays too long, or if the data transfer is complete, the source IMP returns the unused allocation to the destination. With this mechanism we have minimized the inter-message delay and the Hosts can obtain the full bandwidth of the network.

We minimize the delay for a short message by transmitting it to the destination immediately while keeping a copy in the source IMP. If there is space at the destination, it is accepted and passed on to a Host and a RFNM is returned; the source IMP discards the message when it receives the RFNM. If not, the message is discarded, a request for allocation is queued and, when space becomes available, the source IMP is notified that the message may now be retransmitted. Thus, no setup delay is incurred when storage is available at the destination.

The above mechanisms make the IMP network much less sensitive to unresponsive Hosts, since the source Host is effectively held to a transmission rate equal to the reception rate of the destination Host. Further, reassembly lockup is prevented because the destination IMP will never have to turn away a multipacket message destined for one of its Hosts, since reassembly storage has been allocated for each such message in the network.

Source-to-destination sequence control

In addition to its primary function as a flow control mechanism, the link mechanism also originally provided the basis for source-to-destination sequence control. Since only one message was permitted at a time on a link, messages on each link were kept in order; duplicates were detected by the sequence number maintained for each link. In addition, the IMPs marked any message less than 80 bits long as a priority message and gave it special handling to speed it across the network, placing it ahead of long messages on output queues.

The tables associated with the link mechanism in each IMP were large and costly to access. Since the link mechanism was no longer needed for flow control, we felt that a less costly mechanism should be employed for sequence control. We thus decided to eliminate the link mechanism from the IMP subnetwork. RFNMs are still returned to the source Host on a link basis, but link numbers are used only to allow Hosts to identify messages. To replace the per-link sequence control mechanism, we decided upon a sequence control mechanism based on a single logical "pipe" between each source and destination IMP. Each IMP maintains an independent message number sequence for each pipe. A message number is assigned to each message at the source IMP and this message number is checked at the destination IMP. All Hosts at the source and destination IMPs share this message space. Out of an

* This mechanism is similar to that implemented at the level of Host-to-Host protocol,[6,7,8] indicative of the fact that the same sort of problems occur at every level in a communications system.

eight-bit message number space (large enough to accommodate the settling time of the network), both the source and destination keep a small window of currently valid message numbers, which allows several messages to be in the pipe simultaneously. Messages arriving at a destination IMP with out-of-range message numbers are duplicates to be discarded. The window is presently four numbers wide, which seems about right considering the response time required of the network. The message number serves two purposes: it orders the four messages that can be in the pipe, and it allows detection of duplicates. The message number is internal to the IMP subnetwork and is invisible to the Hosts.

A sequence control system based on a single source/destination pipe, however, does not permit priority traffic to go ahead of other traffic. We solved this problem by permitting two pipes between each source and destination, a priority (or low delay) pipe and a nonpriority (or high bandwidth) pipe. To avoid having each IMP maintain two eight-bit message number sequences for every other IMP in the network, we coupled the low delay and high bandwidth pipe so that duplicate detection can be done in common, thus requring only one eleven-bit message number sequence for each IMP.

The eleven-bit number consists of a one-bit priority/non-priority flag, two bits to order priority messages, and eight bits to order all messages. For example, if we use the letters A, B, C, and D to denote the two-bit order numbers for priority messages and the absence of a letter to indicate a nonpriority message, we can describe a typical situation as follows: The source IMP sends out nonpriority message 100, then priority messages 101A and 102B, and then nonpriority message 103. Suppose the destination IMP receives these messages in the order 102B, 101A, 103, 100. It passes these messages to the Host in the order 101A, 102B, 100, 103. Message number 100 could have been sent to the destination Host first if it had arrived at the destination first, but the priority messages are allowed to "leapfrog" ahead of message number 100 since it was delayed in the network. The IMP holds 102B until 101A arrives, as the Host must receive priority message A before it receives priority message B. Likewise, message 100 must be passed to the Host before message 103.

Hosts may, if they choose, have several messages outstanding simultaneously to a given destination but, since priority messages can "leapfrog" ahead, and the last message in a sequence of long messages may be short, priority can no longer be assigned strictly on the basis of message length. Therefore, Hosts must explicitly indicate whether a message has priority or not.

With message numbers and reserved storage to be accurately accounted for, cleaning up in the event of a lost message must be done carefully. The source IMP keeps track of all messages for which a RFNM has not yet been received. When the RFNM is not received for too long (presently about 30 seconds), the source IMP sends a control message to the destination inquiring about the possibility of an incomplete transmission. The destination responds to this message by indicating whether the message in question was previously received or not. The source IMP continues inquiring until it receives a response. This technique guarantees that the source and destination IMPs keep their message number sequences synchronized and that any allocated space will be released in the rare case that a message is lost in the subnetwork because of a machine failure.

IMP-to-IMP transmission control

We have adopted a new technique for IMP-to-IMP transmission control which improves efficiency by 10-20 percent over the original separate acknowledge/timeout/retransmission approach described in Reference 1. In the new scheme, which is also used for the Very Distant Host,[9] and which is similar to Reference 10, each physical network circuit is broken into a number of logical "channels," currently eight in each direction. Acknowledgments are returned "piggy-backed" on normal network traffic in a set of acknowledgment bits, one bit per channel, contained in every packet, thus requiring less bandwidth than our original method of sending each acknowledge in its own packet. The size of this saving is discussed later in the paper. In addition, the period between retransmissions has been made dependent upon the volume of new traffic. Under light loads the network has minimal retransmission delays, and the network automatically adjusts to minimize the interference of retransmissions with new traffic.

Each packet is assigned to an outgoing channel and carries the "odd/even" bit for its channel (which is used to detect duplicate packet transmissions), its channel number, and eight acknowledge bits—one for each channel in the reverse direction.

The transmitting IMP continually cycles through its used channels (those with packets associated with them), transmitting the packets along with the channel number and the associated odd/even bit. At the receiving IMP, if the odd/even bit of the received packet does *not* match the odd/even bit associated with the appropriate receive channel, the packet is accepted and the receive odd/even bit is complemented, otherwise the packet is a duplicate and is discarded.

Every packet arriving over a line contains acknowledges for all eight channels. This is done by copying the *receive* odd/even bits into the positions reserved for the eight acknowledge bits in the control portion of *every* packet transmitted. In the absence of other traffic, the acknowledges are returned in "null packets" in which *only* the acknowledge bits contain relevant information (i.e., the channel number and odd/even bit are meaningless; null packets are not acknowledged). When an IMP receives a packet, it compares (bit by bit) the acknowledge bits against the *transmit* odd/even bits. For each *match* found, the corresponding channel is marked unused, the corresponding packet is discarded, and the transmit odd/even bit is complemented.

In view of the large number of channels, and the delay that is encountered on long lines, some packets may have to wait an inordinately long time for transmission. We do not want a one-character packet to wait for several thousand-bit packets to be transmitted, multiplying by 10 or more the effective delay seen by the source. We have, therefore, instituted the following transmission ordering scheme: priority packets which have never been transmitted are sent first; next sent are any regular packets which have never been transmitted; finally, if there are no new packets to send, previously transmitted packets which are unacknowledged are sent. Of course, unacknowledged packets are periodically retransmitted even when there is a continuous stream of new traffic.

In implementing the new IMP-to-IMP acknowledgment system, we encountered a race problem. The strategy of continuously retransmitting a packet in the absence of other traffic introduced difficulties which were not encountered in the original system, which retransmitted only after a long timeout. If an acknowledgment arrives for a packet which is currently being retransmitted, the output routine must prevent the input routine from freeing the packet. Without these precautions, the header and data in the packet could be changed while the packet was being retransmitted, and all kinds of "impossible" conditions result when this "composite" packet is received at the other end of the line. It took us a long time to find this bug!*

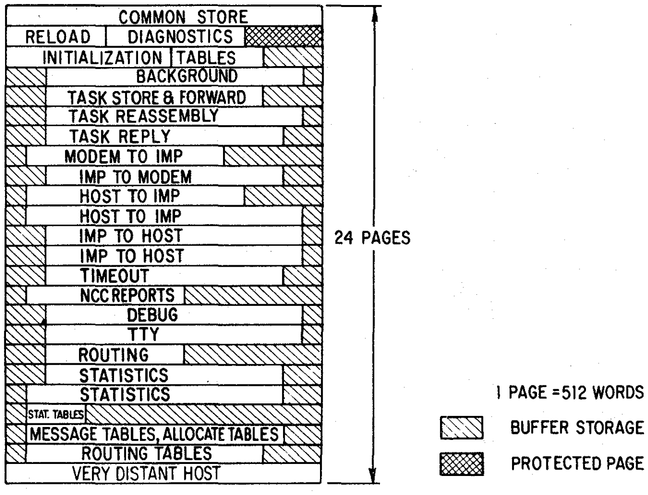


Figure 3—Map of core storage

PROGRAM STRUCTURE

Implementation of the IMPs required the development of a sophisticated computer program. This program was previously described in Reference 1. As stated then, the principal function of the IMP program is the processing of packets, including the following: segmentation of Host messages into packets; receiving, routing, and transmitting of store-and-forward packets; retransmitting unacknowledged packets; reassembling packets into messages for transmission into a Host; and generating RFNMs and other control messages. The program also monitors network status, gathers statistics, and performs on-line testing. The program was originally designed, constructed, and debugged over a period of about one year by three programmers.

Recently, after about two and one-half years of operation in up to twenty-five IMPs throughout the network, the operational program was significantly modified. The modification implemented the algorithms described in the previous sections, thereby eliminating causes of network lockup and improving the performance of the IMP. The modification also extended the capabilities of the IMP so it can now interface to Hosts over common carrier circuits (a Very Distant Host[9]), efficiently manage buffers for lines with a wide range of speeds, and perform better network diagnostics. After prolonged study and preliminary design,[3,4] this program revision was implemented and debugged in about nine man months.

* Interestingly, a similar problem exists on another level, that of source-destination flow control. If an IMP sends a request for allocation, either single- or multi-packet, to a neighboring IMP, it will periodically retransmit it until it receives an acknowledgment. If it receives an allocation in return, it will immediately begin to transmit the first packet of the message. The implementation in the IMP program sends the request from the same buffer as the first packet, merely marking it with a request bit. If an allocation arrives while the request is in the process of being retransmitted, the program must wait until it has been completely transmitted before it sends the same buffer again as the first packet, since the request bit, the odd/even bit, the acknowledge bits, and the message number (for a multipacket request) will be changed. This was another difficult bug.

We shall emphasize in this section the structural changes the program has recently undergone.

Data structures

Figure 3 shows the layout of core storage. As before, the program is broken into functionally distinct pieces, each of which occupies one or two pages of core. Notice that code is generally centered within a page, and there is code on every page of core. This is in contrast to our previous practice of packing code toward the beginning of pages and pages of code toward the beginning of memory. Although the former method results in a large contiguous buffer area near the end of memory, it has breakage at every page boundary. On the other hand, "centering" code in pages such that there are an integral number of buffers between the last word of code on one page and the first word of code on the next page eliminates almost all breakage.

There are currently about forty buffers in the IMP, and the IMP program uses the following set of rules to allocate the available buffers to the various tasks requiring buffers:

- Each line must be able to get its share of buffers for input and output. In particular, one buffer is always allocated for output on each line, guaranteeing that output is always possible for each line; and double buffering is provided for input on each line, which permits all input traffic to be examined by the program, so that acknowledgments can always be processed, which frees buffers.
- An attempt is made to provide enough store-and-forward buffers so that all lines may operate at full capacity. The number of buffers needed depends directly on line distance and line speed. We currently limit each line to eight or less buffers, and a pool is provided for all lines. Some numerical results on line utilization are presented in a later section. Currently, a maximum of twenty buffers is available in the store-and-forward pool.
- Ten buffers are always allocated to reassembly storage, allowing allocations for one multipacket message and two single-packet messages. Additional buffers may be claimed for reassembly, up to a maximum of twenty-six.

Figure 4 summarizes the IMP table storage. All IMPs have identical tables. The IMP program has twelve words of tables for each of the sixty-four IMPs now possible in the network. The program has ninety-one words of tables for each of the eight Hosts (four real and four fake) that can be connected; additionally, twelve words of code are replicated for each real Host that can be connected. The program has fifty-five words of tables for each of the five lines that can be connected; additionally, thirty-seven words of code are replicated for each line that can be connected. The program also has tables for initialization, statistics, trace, and so forth.

The size of the initialization code and the associated tables deserves mention. This was originally quite small. However, as the network has grown and the IMP's capabilities have been expanded, the amount of memory dedicated to initialization has steadily grown. This is mainly due to the fact that the IMPs are no longer identical. An IMP may be required to handle a Very Distant Host, or TIP hardware, or five lines and two Hosts, or four Hosts and three lines, or a very high speed line, or, in the near future, a satellite link. As the physical permutations of the IMP have continued to increase, we have clung to the idea that the program should be identical in all IMPs, allowing an IMP to reload its program from a neighboring IMP and providing other considerable advantages. However, maintaining only one version of the program means that the program must rebuild itself during initialization to be the proper program to handle the particular physical configuration of the IMP. Furthermore, it must be able to turn itself back into its nominal form when it is reloaded into a neighbor. All of this takes tables and code. Unfortunately, we did not foresee the proliferation

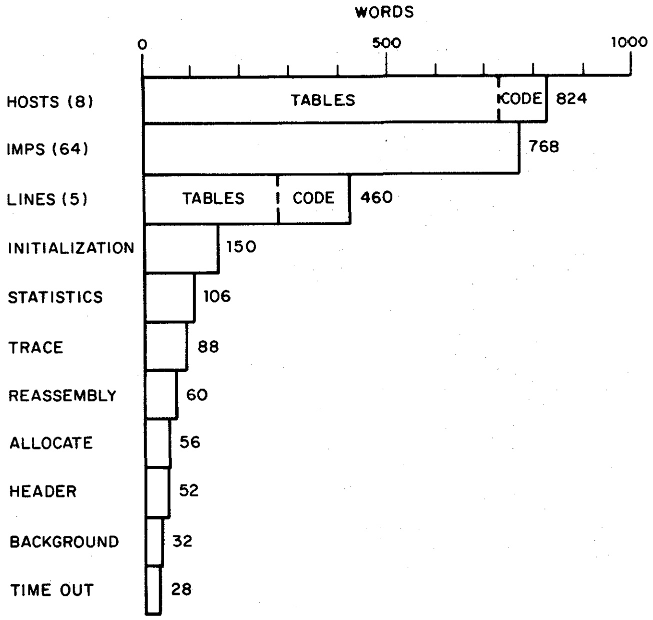


Figure 4—Allocation of IMP table storage

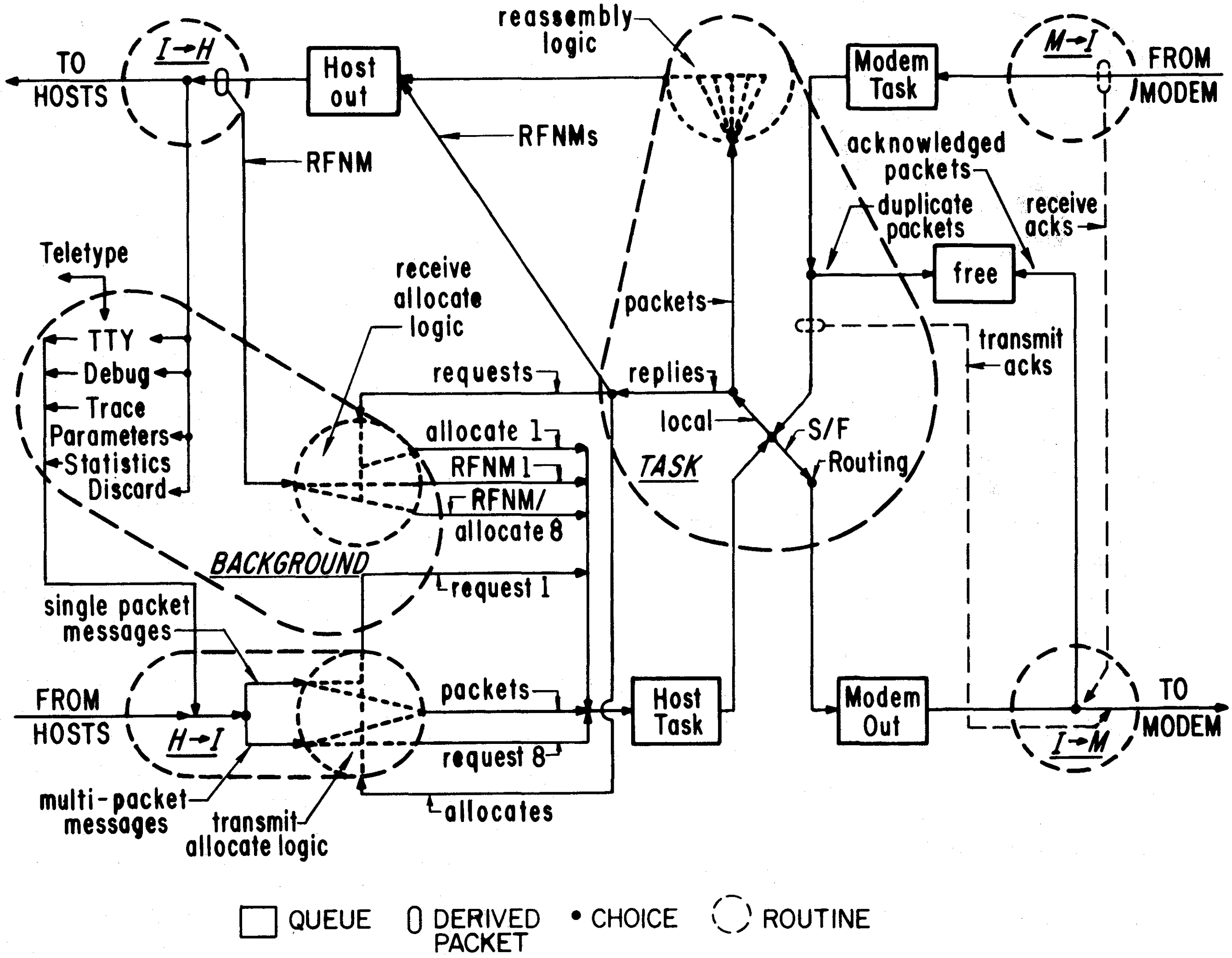


Figure 5—Packet flow and processing

of IMP configurations which has taken place; therefore, we cannot conveniently compute the program differences from a simple configuration key. Instead, we must explicitly table the configuration irregularities.

The packet processing routines

Figure 5 is a schematic drawing of packet flow and packet processing.* We here briefly review the functions of the various packet-processing routines and note important new features.

Host-to-IMP (H→I)

This routine handles messages being transmitted from Hosts at the local site. These Hosts may either be real Hosts or fake Hosts (TTY, Debug, etc.). The routine acquires a message number for each message and passes the message through the transmi allocation logic which requests a reassembly allocation from the destination IMP. Once this allocation is received, the message is broken into packets which are passed to the Task routine via the Host Task queue.

Task

This routine directs packets to their proper destination. Packets for a local Host are passed through the

* Cf. Figure 9 of Reference 1.

reassembly logic. When reassembly is complete, the reassembled message is passed to the IMP-to-Host routine via the Host Out queue. Certain control messages for the local IMP are passed to the transmit or receive allocate logic. Packets to other destinations are placed on a modem output queue as specified by the routing table.

IMP-to-Modem (I→M)

This routine transmits successive packets from the modem output queues and sends piggybacked acknowledgments for packets correctly received by the Modem-to-IMP routine and accepted by the Task routine.

Modem-to-IMP (M→I)

This routine handles inputs from modems and passes correctly received packets to the Task routine via the Modem Task queue. This routine also processes incoming piggybacked acknowledges and causes the buffers for correctly acknowledged packets to be freed.

IMP-to-Host (I→H)

This routine passes messages to local Hosts and informs the background routine when a RFNM should be returned to the source Host.

Background

The function of this routine includes handling the IMP's console Teletype, a debugging program, the statistics programs, the trace program, and several routines which generate control messages. The programs which perform the first four functions run as fake Hosts (as described in Reference 1). These routines simulate the operation of the Host/IMP data channel hardware so the Host-to-IMP and IMP-to-Host routines are unaware they are communicating with anything other than a real Host. This trick saved a large amount of code and we have come to use it more and more. The programs which send incomplete transmission messages, send and return allocations, and send RFNMs also reside in the background program. However, these programs run in a slightly different manner than the fake Hosts in that they do not simulate the Host/IMP channel hardware. In fact, they do not go through the Host/IMP code at all, but rather put their messages directly on the task queue. Nonetheless, the principle is the same.

Timeout

This routine, which is not shown in Figure 5, performs a number of periodic functions. One of these functions is garbage collection. Every table, most queues, and many states of the program are timed out. Thus, if an entry remains in a table abnormally long or if a routine remains in a particular state for abnormally long, this entry or state is garbage-collected and the table or routine is returned to its initial or nominal state. In this way, abnormal conditions are not allowed to hang up the system indefinitely.

The method frequently used by the Timeout routine to scan a table is interesting. Suppose, for example, every entry in a sixty-four entry table must be looked at every now and then. Timeout could wait the proper interval and then look at every entry in the table on one pass. However, this would cause a severe transient in the timing of the IMP program as a whole. Instead, one entry is looked at each time through the Timeout routine. This takes a little more total time but is much less disturbing to the program as a whole. In particular, worst case timing problems (for instance, the processing time between the end of one modem input and the beginning of the next) are significantly reduced by this technique. A particular example of the use of this technique is with the transmission of routing information to the IMP's neighbors. In general, an IMP can have five neighbors. Therefore, it sends routing information to one of its neighbors every 125 msec rather than to all of its neighbors every 625 msec.

In addition to timing out various states of the program, the Timeout routine is used to awaken routines which have put themselves to sleep for a specified period. Typically these routines are waiting or some resource to become available, and are written as co-routines with the Timeout routine. When they are restarted by Timeout the test is made for the availability of the resource, followed by another delay if the resource is not yet available.

PERFORMANCE EVALUATION

In view of the extensive modifications described in the preceding sections, it was appropriate to recalculate the IMP's performance capabilities. The following section presents the results of the reevaluation of the IMP's performance and comparisons with the performance reports of Reference 1.

Throughput vs. message length

In this section we recalculate two measures of IMP performance previously calculated in Reference 1, the maximum throughput and line traffic. Throughput is the number of Host data bits that traverse an IMP each second. Line traffic is the number of bits that an IMP transmits on its communication circuits per second and includes the overhead of RFNMs, packet headers, acknowledges, framing characters, and checksum characters.

To calculate the IMP's maximum line traffic and throughput, we first calculate the computational load placed on the IMP by the processing of one message. The computational load is the sum of the machine instruction cycles plus the input/output cycles required to process all the packets of a message and their acknowledgments, and the message's RFNM and its acknowledgment. For simplicity in computing the computational load, we ignore the processing required to send and receive the message from a Host since this is only seen by the source and destination IMPs.

A packet has D bits of data, S bits of software overhead, and H bits of hardware overhead. For the original and modified IMP systems, the values of D, S, and H are:

	Original	*Modified*
D	0-1008 bits	0-1008 bits
S	64(packet)+80(ack)= 144 bits	80 bits(packet+ack)
H	72(packet)+72(ack)= 144 bits	72 bits(packet+ack)

The input/output processing time for a packet is the time taken to transfer $D+S$ bits from memory to the modem interface at one IMP plus the time to transfer $D+S$ bits into memory at the other IMP. If R is the input/output transfer rate in bits per second,* then the input/output transfer time for a packet is $2(D+S)/R$. Therefore, the total input/output time, I_m, for P packets in a B bit message is $2(B+P\times S)/R$. The input/output transfer time, I_r, for a RFNM is $2S/R$.

To each of these numbers we must add the program processing time, C; this is about the same for a packet of a message and a RFNM.

For the original IMP program, the program processing time per packet consisted of the following:

Modem Output	100 cycles	Send out packet
Modem Input	100 cycles	Receive packet at other IMP
Task	150 cycles	Process it (route onto an output line)
Modem Output	100 cycles	Send back an acknowledgment
Modem Input	100 cycles	Receive acknowledgment at first IMP
Task	150 cycles	Process acknowledgment
	700 cycles	Program processing time per packet

For the modified IMP program, the program processing time consists of:

Modem Output	150 cycles	Send out packet and piggyback acks
Modem Input	150 cycles	Receive packet and process acks
Task	250 cycles	Process packet
	550 cycles	Program processing time per packet

Finally, we add a percentage, V, for overhead for the various periodic processes in the IMP (primarily the routing computation) which take processor bandwidth. V is presently about 5 percent.

We are now in a position to calculate the computational load (in seconds), L, of one P packet message:

$$L=\underbrace{(P\times C+I_m)\times(1+V)}_{\text{packets}}+\underbrace{(C+I_r)\times(1+V)}_{\text{RFNM}}$$

The maximum throughput, T, is the number of data bits in a single message divided by the computational loads of the message; that is, $T=B/L$.

The maximum line traffic (in bits per second), R, is the throughput plus the overhead bits for the packets of the message and the RFNM divided by the computational load of the message. That is,

$$R=T+\frac{(P+1)\times(S+H)}{L}=\frac{B+(P+1)\times(S+H)}{L}$$

The maximum throughput and line traffic are plotted for various message lengths in Figure 6 for the original and modified programs and for the 516 IMP and the 316 IMP.

* In this calculation we will be making the distinction between the 516 IMP (used originally and reported on in Reference 1) and the 316 IMP (used for all new IMPs). The 516 has a memory cycle time of 0.96 μsec, and the 316 has a cycle of 1.6 μsec. The 316 provides a two-cycle data break, in comparison with the four-cycle data break on the 516. Thus, the input/output transfer rates are 16 bits per 3.84 μsec for the 516 and 16 bits per 3.2 μsec for the 316.

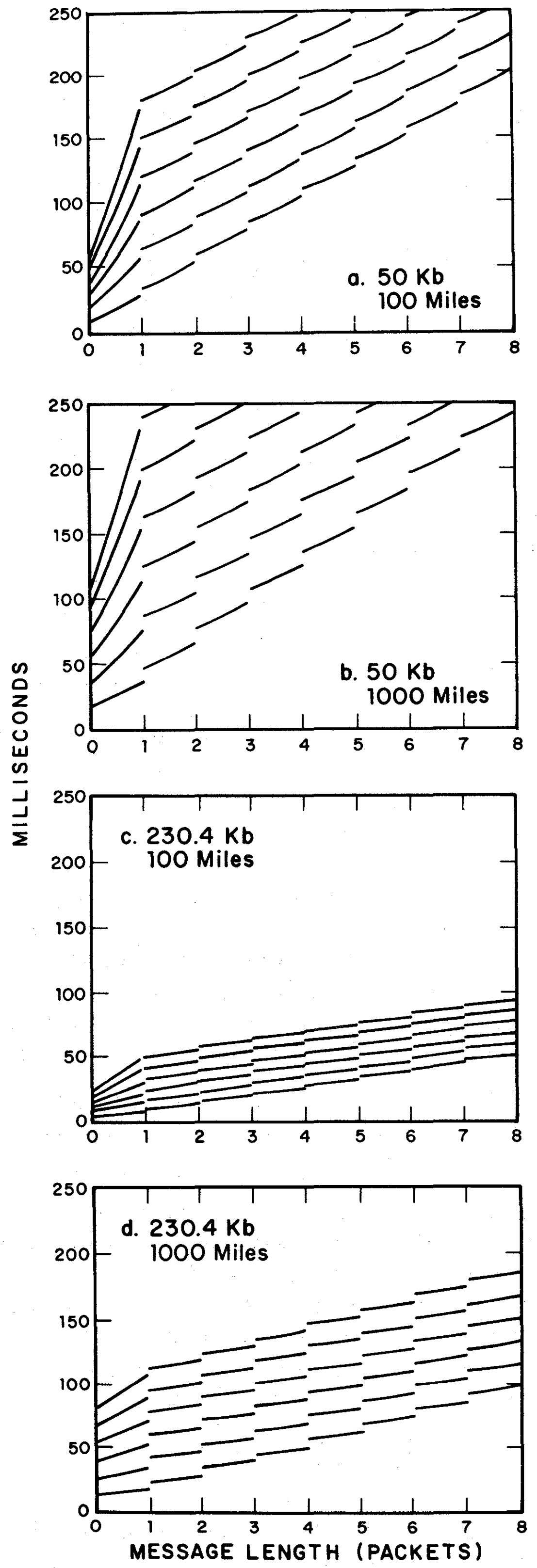


Figure 6—Line traffic and throughput vs. message length. The upper curves plot maximum line traffic, the lower curves plot maximum throughput

The changes to the IMP system can be summarized as follows:

- The program processing time for a store-and-forward packet has been decreased by 20 percent.
- The line throughput has been increased by 4 percent for a 516 IMP and by 7 percent for a 316 IMP.

As a result, the net throughput rate has been increased by 17 percent for a 516 IMP and by 21 percent for a 316 IMP. Thus, a 316 IMP can now process almost as much traffic as a 516 IMP could with the old program. A 516 IMP can now process approximately 850 Kbs.

- The line overhead on a full-length packet has been decreased from 29 percent to 16 percent.

As a result, the effective capacity of the telephone circuits has been increased from thirty-eight full packet messages per second on a 50 Kbs line to forty-three full packet messages per second.

Round trip delay vs. message length

In this section we compute the minimum round trip delay encountered by a message. We define round trip delay as in Reference 1; that is, the delay until the message's RFNM arrives back at the destination IMP. A message has P packets and travels over H hops. The first packet encounters delay due to the packet processing time, C; the transmission delay, T_P; and the propagation delay, L. Each successive packet of the message follows $C+T_P$ behind the previous packet. Since the message's RFNM is a single packet message with a transmission delay, T_R, we can write the total delay as

$$\underbrace{H\times(C+T_P+L)}_{\text{first packet}}+\underbrace{(P-1)\times(C+T_P)}_{\text{successive packets}}+\underbrace{H\times(C+T_R+L)}_{\text{RFNM}}$$

For single packet messages, this reduces to

$$2H(C+L)+H(T_P+T_R)$$

The curves of Figure 7 show minimum round-trip delay through the network for a range of message lengths and hop numbers, and for two sets of line speeds and line lengths. These curves agree with experimental data.[11,12]

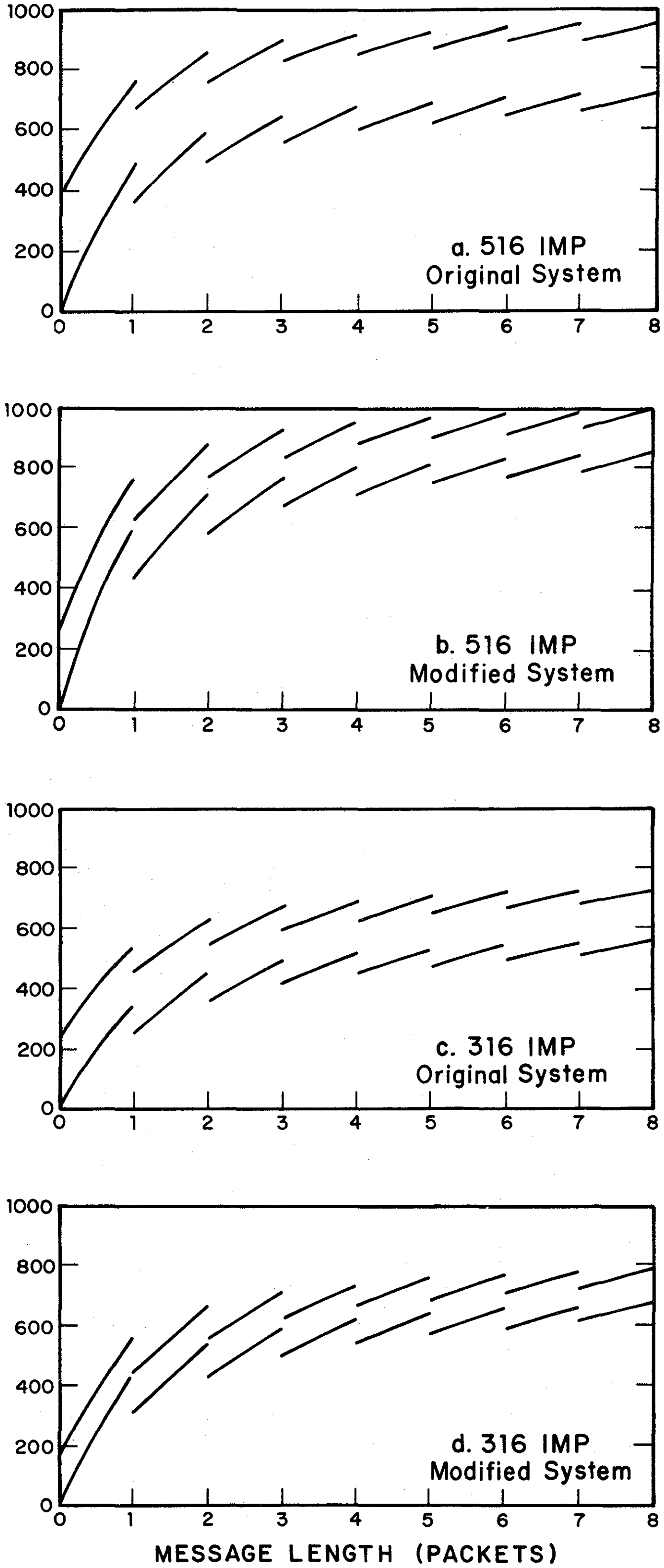


Figure 7—Minimum round trip delay vs. message length. Curves show delay for 1-6 hops

Line utilization

The number of buffers required to keep a communications circuit fully loaded is a function not only of line bandwidth and distance but also of packet length, IMP delay, and acknowledgment strategy. In order to compute the buffering needed to keep a line busy, we need to know the length of time the sending IMP must wait between sending out a packet and receiving an acknowledgment for it. If we assume no line errors, this time is the sum of: propagation delays for the packet and its acknowledgment, P_P and P_A; transmission delays for the packet and its acknowledgment, T_P and T_A; and the IMP processing delay before the acknowledgment is sent. Thus, the number of buffers needed to fully utilize a line is $(P_P+T_P+L+P_A+T_A)/T_P$.

Since $P_P=P_A$, the expression for the number of buffers can be rewritten:

$$\frac{2P}{T_P}+1+\frac{L+T_A}{T_P}$$

That is, the number of buffers needed to keep a line full is proportional to the length of the line and its speed, and inversely proportional to the packet size, with the addition of a constant term.

To compute T_P, we must take into account the mix of short and long packets. Thus, we write

$$T_P=\frac{xT_S+yT_L}{x+y}$$

where x to y is the ratio of number of short packets to number of long packets and T_S and T_L are the transmission delays incurred by short and long packets, respectively. The shortest packet permitted is 152 bits long (entirely overhead); the longest packet is 1160 bits long. Computing T_S and T_L for any given line bandwidth is a simple matter; they typically range from 106 μsec for T_S on a 1.4 Mbs line to 120.5 msec for T_L on a 9.6 Kbs line.

Assuming worst case IMP processing delay (that is, the acknowledge becomes ready for transmission just as the first bit of a maximum length packet is sent), $L=T_L$.

The acknowledge returns in the next outgoing packet at the other IMP, which we assume is of "average" size:*

$$T_A=\frac{T_S+T_L}{2}$$

Propagation delay, P, is essentially just "speed of

* Variations of this assumption have only second order effects on the computation of the number of buffers required.

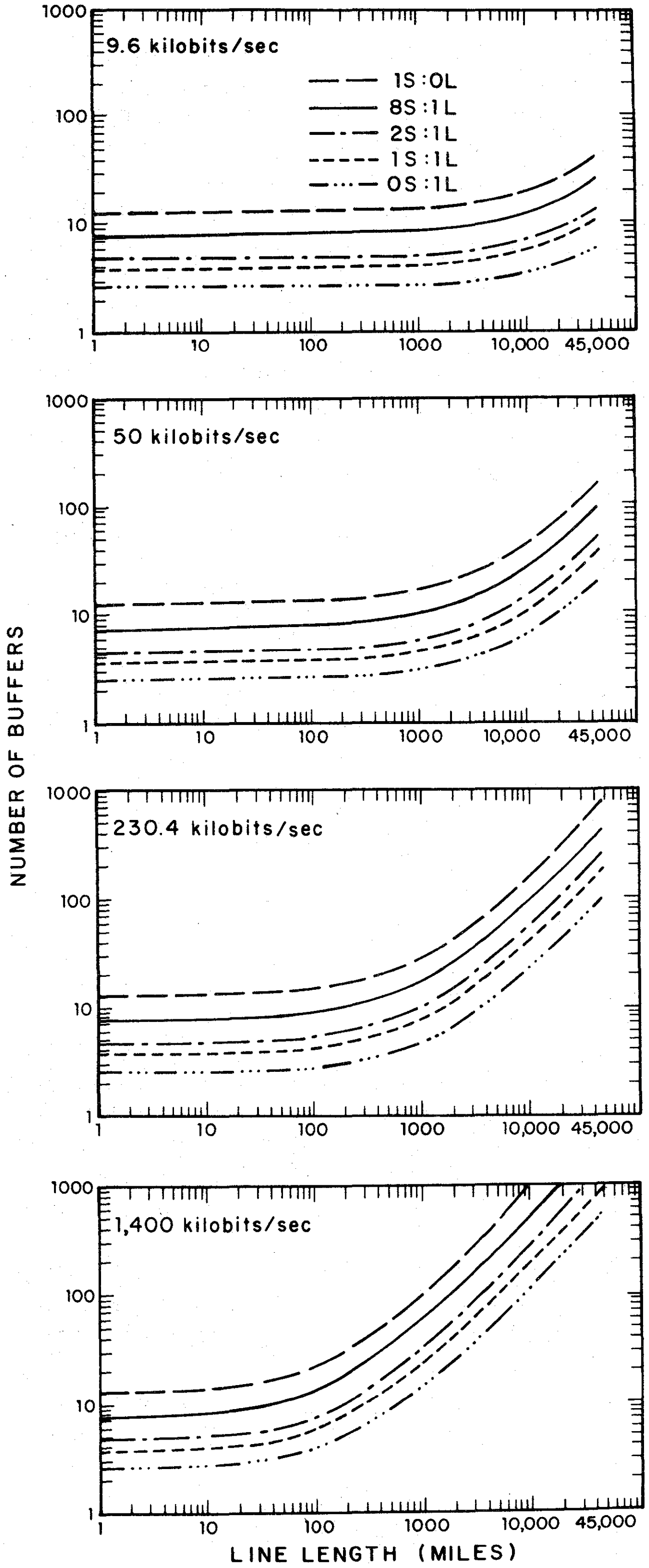


Figure 8—Number of buffers for full line utilization. Traffic mixes are shown as the ratio of number of short packets (S) to number of long packets (L)

light" delay, and ranges from 50 μsec for short lines, through 20 msec for a cross country line, to 275 msec for a satellite link.

We can now compute the number of buffers required to fully utilize a line for any line speed, line length, and traffic mix. Figure 8 gives the result for typical speeds, lengths, and mixes. Note that the knee of the curves occurs at progressively shorter d'stances with increasing line speeds. The constant term dominates the 9.6 Kbs case, and it is almost insignificant for the 1.4 Mbs case. Note also that the separation between members of each family of curves remains constant on the log scale, indicating greatly increased variations with distance.

GENERAL COMMENTS

The ARPA Network has represented a fundamental development in the intersection of computers and communications. Many derivative activities are proceeding with considerable energy, and we list here some of the important directions:

- The present network is expanding, adding IMP and TIP nodes at rates approaching two per month. Other government agencies are initiating efforts to use the network, and increasing rates of growth are likely. As befits the growing operational character of the ARPA Network, ARPA is making efforts to transfer the network from under ARPA's research and development auspices to an operational agency or a specialized carrier of some sort.
- Technical improvements in the existing network are continuing. Arrangements have now been made to permit Host-IMP connections at distances over 2000 feet by use of common-carrier circuits. Arrangements are being made to allow the connection of remote-job-entry terminals to a TIP. In the software area, the routing algorithms are still inadequate at heavy load levels, and further changes in these algorithms are in progress. A major effort is under way to develop an IMP which can cope with megabit/second circuits and higher terminal throughput. This new "high speed modular IMP" will be based on a minicomputer, multiprocessor design; a prototype will be completed in 1973.
- The network is being expanded to include satellite links to oversea nodes, and an entirely new approach is being investigated for the "multi-access" use of satellite channels by message switched digital communication systems.[13] This work could

lead to major changes in world-wide digital communications.

- Many similar networks are being designed by other groups, both in the United States and in other countries. These groups are reviewing the myriad detailed design choices that must be made in the design of message switched systems, and a wide understanding of such networks is growing.
- The existence of the ARPA Network is encouraging a serious review of approaches to obtaining new computer resources. It is now possible to consider investing in major resources, because a national, or even international, network clientele is available over which to amortize the cost of such major resources.
- Perhaps most important, the network has catalyzed important computer research into how programs and operating systems should communicate with each other, and this research will hopefully lead to improved use of all computers.

The ARPA Network has been an exciting development, and there is much yet left to learn.

ACKNOWLEDGMENTS

Dr. Lawrence G. Roberts and others in the ARPA office have been a continuing source of encouragement and support. The entire "IMP group" at Bolt Beranek and Newman Inc. has participated in the development, installation, test, and maintenance of the IMP subnetwork. In addition, Dr. Robert E. Kahn of Bolt Beranek and Newman Inc. was deeply involved in the isolation of certain network weaknesses and in the formative stages of the corrective algorithms. Alex McKenzie made many useful suggestions during the writing of this paper. Linda Ebersole helped with the production of the manuscript.

REFERENCES

1 F E HEART R E KAHN S M ORNSTEIN W R CROWTHER D C WALDEN
The interface message processor for the ARPA computer network
Proceedings of AFIPS 1970 Spring Joint Computer Conference Vol 36 pp 551-567

2 S M ORNSTEIN F E HEART W R CROWTHER H K RISING S B RUSSELL A MICHEL
The terminal IMP for the ARPA computer network
Proceedings of AFIPS 1972 Spring Joint Computer Conference Vol 40 pp 243-254

3 R E KAHN W R CROWTHER
A study of the ARPA network design and performance
Report No 2161 Bolt Beranek and Newman Inc August 1971

4 R E KAHN W R CROWTHER
Flow control in a resource sharing computer network
Proceedings of the Second ACM IEEE Symposium on Problems in the Optimization of Data Communications Systems Palo Alto California October 1971 pp 108-116

5 F HEART S M ORNSTEIN
Software and logic design interaction in computer networks
Infotech Computer State of the Art Report No 6 Computer Networks 1971

6 S CARR S CROCKER V CERF
Host/host protocol in the ARPA network
Proceedings of AFIPS 1970 Spring Joint Computer Conference Vol 36 pp 589-597

7 S CROCKER J HEAFNER R METCALFE J POSTEL
Function-oriented protocols for the ARPA network
Proceedings of AFIPS 1972 Spring Joint Computer Conference Vol 40 pp 271-280

8 A McKENZIE
Host/host protocol for the ARPA network
Available from the Network Information Center as NIC 8246 at Stanford Research Institute Menlo Park California 94025

9 *Specifications for the interconnection of a host and an IMP*
Bolt Beranek and Newman Inc Report No 1822 revised April 1972

10 K BARTLETT R SCANTLEBURY P WILKINSON
A note on reliable full-duplex transmission over half duplex links
Communications of the ACM 12 5 May 1969 pp 260-261

11 G D COLE
Computer networks measurements techniques and experiments
UCLA-ENG-7165 Computer Science Department School of Engineering and Applied Science University of California at Los Angeles October 1971

12 G D COLE
Performance measurements on the ARPA computer network
Proceedings of the Second ACM IEEE Symposium on Problems in the Optimization of Data Communications Systems Palo Alto California October 1971 pp 39-45

13 N ABRAMSON
The ALOHA system—Another alternative for computer communications
Proceedings of AFIPS 1970 Fall Joint Computer Conference Vol 37 pp 281-285

SUPPLEMENTARY BIBLIOGRAPHY

(The following describe issues related to, but not directly concerned with, those discussed in the text.)

H FRANK I T FRISCH W CHOU
Topological considerations in the design of the ARPA computer network
Proceedings of AFIPS 1970 Spring Joint Computer Conference Vol 36 pp 581-587

H FRANK R E KAHN L KLEINROCK
Computer communication network design—Experience with theory and practice
Proceedings of AFIPS 1972 Spring Joint Computer Conference Vol 40 pp 255-270

R E KAHN
Terminal access to the ARPA computer network
Courant Computer Symposium 3—Computer Networks
Courant Institute New York November 1970

L KLEINROCK
Analytic and simulation methods in computer network design
Proceedings of AFIPS 1970 Spring Joint Computer Conference
Vol 36 pp 569-579

A A McKENZIE B P COSELL J M McQUILLAN
M J THROPE
The network control center for the ARPA network
To be presented at the International Conference on Computer Communications Washington D C October 1972

L G ROBERTS
Extension of packet communication technology to a hand-held personal terminal
Proceedings of AFIPS 1972 Spring Joint Computer Conference
Vol 40 pp 295-298

L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
Proceedings of AFIPS 1970 Spring Joint Computer Conference
Vol 36 pp 543-549

R THOMAS D A HENDERSON
McROSS—A multi-computer programming system
Proceedings of AFIPS 1972 Spring Joint Computer Conference
Vol 40 pp 281-294

Cost effective priority assignment in network computers

by E. K. BOWDON, SR.

University of Illinois
Urbana, Illinois

and

W. J. BARR

Bell Telephone Laboratories
Piscataway, New Jersey

INTRODUCTION

Previously, the study of network computers has been focused on the analysis of communication costs, optimal message routing, and the construction of a communications network connecting geographically distributed computing centers. While these problems are far from being completely solved, enough progress has been made to allow the construction of reasonably efficient network computers. One problem which has not been solved, however, is making such networks economically viable. The solution of this problem is the object of our analysis.

With the technological problems virtually solved, it is readily becoming apparent that no matter whose point of view one takes, the only economically justifiable motivation for building a network computer is resource sharing. However, the businessmen, the users, the people with money to spend, could not care less whose resources they are using for their computer runs. They care only that they receive the best possible service at the lowest possible price. The computing center manager who cannot fill this order will soon find himself out of customers.

"The best possible service..." is, in itself, a tall order to fill. The computing center manager finds himself in a position to offer basically two kinds of computing services: contract services and demand services. Contract services are those services which the manager agrees to furnish, at a predetermined price, within specified time periods. Examples of this type of service include payroll runs, billings, and inventory updates. Each of these is run periodically and the value placed by the businessman on the timely completion of the task is large. Demand services are those services which, though defined in advance, may be required at any time and at a possibly unknown price. Frequently the only previous agreements made refer to the type of service to be delivered and limits on how much will be demanded. Examples of tasks which are run on a demand basis include research, information requests, and program debugging runs. University computing centers generally find that most of the services which they offer are of this type.

Every installation manager who offers either contract or demand services should have a solid and acceptable answer to the critical question "What do I do when my computer breaks down?" If he wishes to ensure that he can meet all commitments, the only answer is to transfer tasks to another processor. This is where network computers enter the picture. If the center is part of a network computer, tasks can easily and quickly be transferred to another center for processing. The concept of transferring tasks between centers through a broker has been widely discussed in the literature.[1,2]

Our basic assumption is that economic viability for network computers is predicated on efficient resource sharing. This was, in fact, a major reason for the construction of several networks—to create the capability of using someone else's special purpose machine or unique process without having to physically transport the work. This type of resource sharing is easily implemented and considerable work has been done toward this goal. There is, however, another aspect of resource sharing which has not been studied thoroughly: load-leveling. By load-leveling we mean the transfer of tasks between computing centers for the purpose of improving the throughput of the network or other criteria. We contend that the analysis and implementation of user-oriented load-leveling is the key to developing economically self-supporting network computers.

A SCENARIO OF COST EFFECTIVENESS

Until recently, efforts to measure computer efficiency have centered on the measurement of resource (including processor) idle time. A major problem with this philosophy is that it assumes that all tasks are of roughly equal value to the user and hence the operation of the system.

As an alternative to the methods used in the past, we propose a priority assignment technique designed to represent the worth of tasks in the system. We present the hypothesis that tasks requiring equivalent use of resources are not necessarily of equivalent worth to the user with respect to time. We would allow the option for the user to specify a "deadline" after which the value of his task would decrease, at a rate which he can specify, to a system determined minimum. With this in mind, we have proposed a measure of cost effectiveness with which we can evaluate the performance of a network with an arbitrary number of interconnected systems, as well as each system individually.[3]

We define our measure of cost effectiveness γ, as follows:

$$\gamma=(Lq/M)^{\alpha}\left[1-(M-Lq)\bigg/\sum_{i=0}^{R-1}\beta(i)\right] \qquad (1)$$

where

Lq is the number of tasks in the queue,
M is the maximum queue length,
R is the number of priority classes,
α is a measure (system-determined) of the "dedicatedness" of the CPU to the processing of tasks in the queue,

and

$$\beta(i)=(R-i)\sum_{j=1}^{n}[g(j)/f(j)]$$

where

$g(j)$ is the reward for completing task j (a user specified function of time),

and

$f(j)$ is the cost (system determined) to complete task j.

The term $(Lq/M)^{\alpha}$ is a measure of the relevance of the queue to processing activities. Similarly, we can look at $\beta(i)$ as a measure of resource utilization. Note that $\beta(i)$ indicates a ratio of reward to cost for a given priority class and is sensitive to the needs of the user and the requirements imposed on the installation. It is user sensitive because the user specifies the reward and is installation sensitive because the cost for processing a task is determined by the system. The measure of CPU dedicatedness (α), on the other hand, is an entirely installation sensitive parameter.

The first problem which becomes apparent is that which arises if

$$\sum_{i=0}^{R-1}\beta(i)=0.$$

This occurs only in the situation where there is exactly one priority class (i.e., the non-priority case). We will finesse away this problem by defining

$$(M-Lq)\bigg/\sum_{i=0}^{R-1}\beta(i)=0$$

for this case. Intuitively, this is obvious, since the smaller this term gets, the more efficiently (in terms of reward) a system is using its resources. Furthermore, in the absence of priorities, the order in which tasks are executed is fixed, so this term becomes irrelevant to our measure of cost effectiveness. Thus, for the non-priority case, we have

$$\gamma=(Lq/M)^{\alpha}$$

which is simply the measure of the relevance of the queue to processing activities. This is precisely what we want if we are going to consider only load-leveling in non-priority systems. However, we are interested in the more general case in which we can assign priorities.

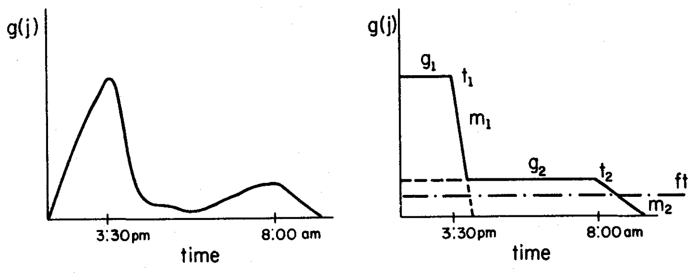
An estimate of the cost to complete task j, $f(j)$ is readily determined from the user-supplied parameters requesting resources. Frequently these estimated parameters are used as upper limits in resource allocation and the operating system will not allow the program to exceed them. As a result, the estimates tend to be high. On the other hand, lower priorities are usually assigned to tasks requiring a large amount of resources. So the net effect is that the user's parameters reflect his best estimate and we may be reasonably confident that they truly reflect his needs.

At the University of Illinois computing center, for example, as of July 26, 1971, program charges are estimated by the following formula:

$$\text{cents}=a(X+Y)(bZ+c)+d \qquad (2)$$

where

X = CPU time in centiseconds,
Y = number of I/O requests,



(a) Ideal function.

(b) Approximate function.

Figure 1—Example of user's reward function

Z = core size in kilobytes,

a, b, c are weighting factors currently having the values 0.04, 0.0045, and 0.5, respectively.

and

d is an extra charge factor including $1.00 cover charge plus any special resources used (tape/disk storage, card read, cards punched, plotter, etc.).

The main significance of the reward function $g(j)$ specified by the user is that it allows us to determine a deadline or deadlines for the task. Typically we might expect $g(j)$ to be a polynomial in t, where t is the time in the system. For example, the following thoughts might run through the user's head: "Let's see, it's 10:00 a.m. now and I don't really need immediate results since I have other things to do. However, I do need the output before the 4:00 p.m. meeting. Therefore, I will make 3:30 p.m. a primary deadline. If it isn't done before the meeting, I can't use the results before tomorrow morning, so I will make 8:00 a.m. a secondary deadline. If it isn't done by then I can't use the results, so after 8:00 a.m. I don't care."

The function $g(j)$ this user is thinking about would probably look something like Figure 1a. Now, this type of function poses a problem in that it is difficult for the user to specify accurately and would require an appreciable amount of overhead to remember and compute. Notice, however, that even if turnaround time is immediate, the profit oriented installation manager would put the completed task on a shelf (presumably an inexpensive storage device) and not give it to the user until just before the deadline—thus collecting the maximum reward. As a result, there is little reason for specifying anything more than the deadlines, the rewards associated with meeting the deadlines, and the rate of decrease of the reward between deadlines, if any. Applying this reasoning to Figure 1a we obtain Figure 1b. Note that this function is completely specified with only six parameters (deadlines t_1, t_2; rewards g_1, g_2; and rates of decrease m_1, m_2).

In general, we may assume that $g(j)$ is a monotonically non-increasing, piecewise linear, reward function consisting of n distinct sets of deadlines, rewards, and rates of decrease. Thus we can simply specify $g(j)$ with $3n$ parameters where n is the number of deadlines specified.

Note that, in effect, the user specifies an abort time when the $g(j)$ he specifies becomes less than $f(j)$. If the installation happens to provide a "lower cost" service, $\tilde{f}(j)$ and if $g(j) > \tilde{f}(j)$, this task would be processed, but only when all the tasks with higher $g(j)$ had been processed.

Now, what we are really interested in, is not so much an absolute reward, but a ratio of reward to cost. Since $f(j)$ is, at best, only an estimate of cost, we cannot reasonably require a user to specify an absolute reward. A more equitable arrangement would be to specify the rewards in terms of a ratio $g(j)/f(j)$ associated with each deadline. This ratio is more indicative of the relative worth of a task, both to the system and to the user, since it indicates the return on an investment.

PRIORITY ASSIGNMENT

Let us now turn our attention to the development of a priority assignment scheme which utilizes the reward/cost ratios described in the previous section. We begin by quantizing the continuum of reward/cost ratios into R distinct intervals. Each of these intervals is then assigned to one of R priority classes $0, 1, 2, \ldots, R-1$ with priority 0 being reserved for tasks with highest reward/cost ratios and priority $R-1$ for tasks with reward/cost ratios of unity or less. A task entering the system will be assigned a priority according to its associated reward/cost ratio.

We want to guarantee, if possible, that all priority 0 tasks will meet their deadlines. Furthermore, if all priority 0 tasks can meet their deadlines, we want to guarantee, if possible, that all priority 1 tasks will meet their deadlines and, in general, if all priority k tasks can meet their deadlines, we want to guarantee that as many priority class $k+1$ tasks as possible will meet their deadlines. (Note that we are concerned with guaranteeing deadlines rather than rationing critical resources.)

To facilitate the priority assignment, we introduce the following notation: For priority k, let T_i denote the i^{th} task. Then we assume for each T_i that we receive the following information vector:

$$(T_i, g/f, d_i, \tau_i, s_i)$$

where

T_i is an identifier,

g/f is the reward/cost ratio associated with meeting the task's deadline,

d_i is the task's deadline associated with g/f,

τ_i is the maximum processing time for the task,

and

$s_i = d_i - \tau_i$, is the latest time at which the task may start processing and still be assured of meeting its deadline.

Now since each task has an associated deadline and maximum processing time, we can use the resulting latest start time as the basis for assigning positions to tasks within a priority class. A last come, first served rule will be used to break ties. Additionally, we will use a compacting scheme to ensure that as many tasks as possible start processing before their latest start times. More formally our algorithm may be stated as follows:

Priority assignment algorithm

First, we assign a new task a priority based on its g/f ratio, say priority k. Then within class k, its position is determined as follows:

1. Beginning with the last priority k task (that is, the task with latest deadline) search forward until two tasks, T_{j-1} and T_j, are found such that $d_{j-1} < d_i \leq d_j$. Insert T_i between the two tasks, assign it a start time $s_i = d_i - \tau_i$, and renumber the tasks behind T_j accordingly (i.e., T_i becomes T_j, T_j becomes T_{j+1}, etc.).
2. Now, if $s_{j-1} + \tau_{j-1} \leq s_j \leq s_{j+1} - \tau_j$ there is sufficient float time between T_{j-1} and T_{j+1} for T_j to be processed on time and the priority assignment is complete. However, if either $s_{j-1} + \tau_{j-1} > s_j$ or $s_j + \tau_j > s_{j+1}$, a deadline might be missed; so we proceed with Step 3.
3. **Compacting scheme** Let f_j. denote the float time between any two tasks T_{j-1} and T_j, where f_j is defined:

   $$f_j = s_j - (s_{j-1} + \tau_{j-1})$$

   Then, F_j, the total float time preceding T_j, is given by:

   $$F_j = \sum_{k=1}^{j} f_j = s_j - t + \sum_{k=1}^{j} \tau_j \qquad (3)$$

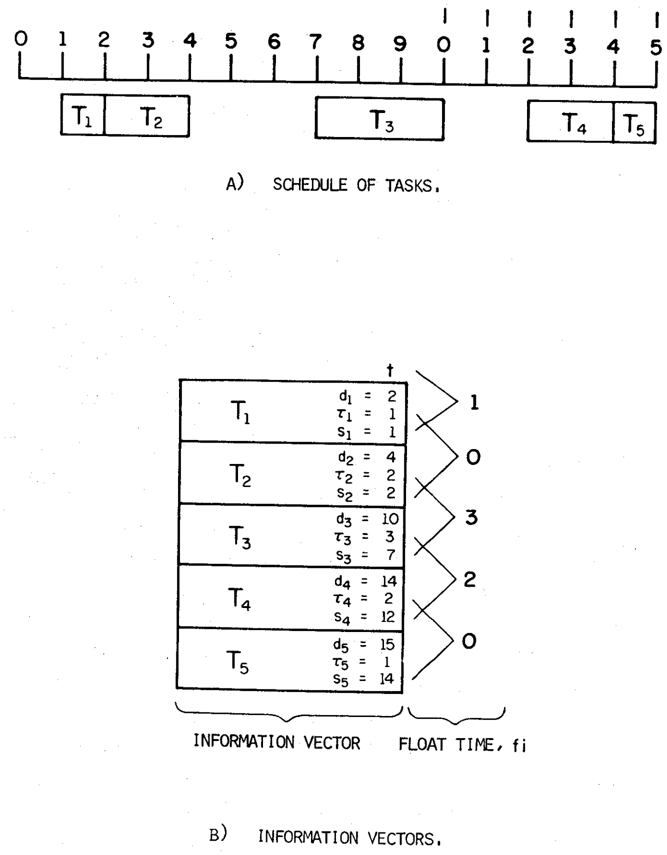
   where t is the current time. Now, starting with task T_j, if $s_j + \tau_j > s_{j+1}$ and $F_j \geq s_j + \tau_j - s_{j+1}$ we assign a new starting time to T_j given by:

   $$s_j = s_{j+1} - \tau_j$$

   and we continue with T_{j-1}, T_{j-2}, etc., until we encounter a task T_k, $k \neq j$, such that $s_k \leq s_{k+1} - \tau_k$. (Note that T_{j+1} and all its predecessors are guaranteed to meet their deadlines.)
4. However, if $s_j + \tau_j > s_{j+1}$ but $F_j < s_j + \tau_j - s_{j+1}$, we do not assign a new start time to T_j. Instead we leave the start time at its latest critical value, even though it may not start processing at that time. We observe that many tasks may not require all of the processing time specified by their maxima, and as a result sufficient float time may be created later to enable the task, T_j, to meet its deadline.

Examples

Several examples will now be given to illustrate the efficacy of the algorithm. Suppose we have determined that a task, T_i, should have priority k and that at time



A) SCHEDULE OF TASKS.

B) INFORMATION VECTORS.

Figure 2—State of priority Class k at time $t = 0$

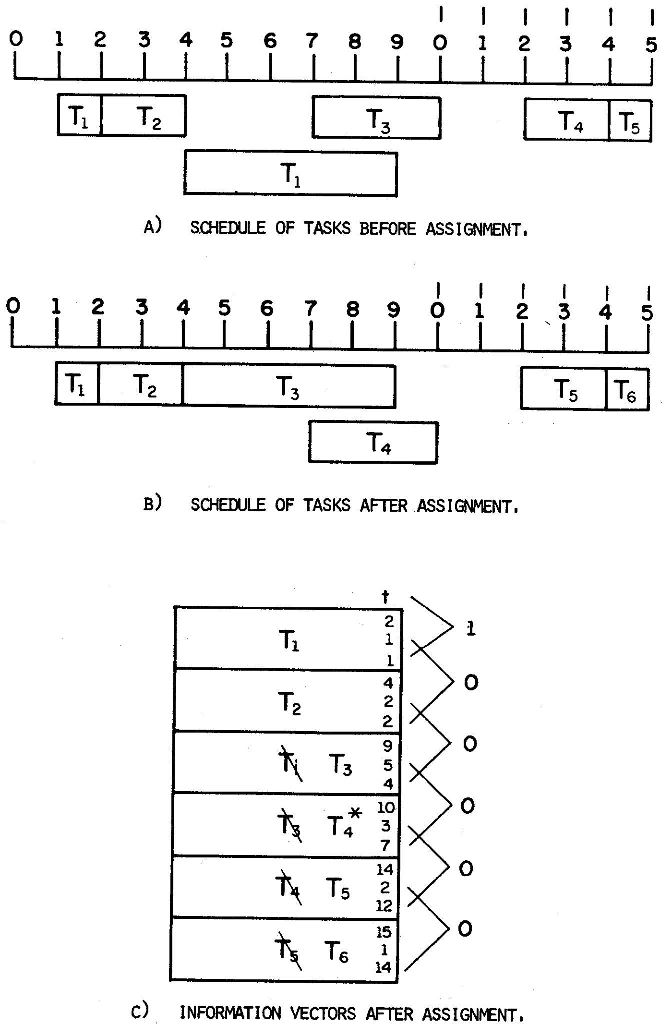


Figure 3—Results of priority assignments for Example (i)

$t=0$, the state of priority class k is that shown in Figure 2. (Note that since all priority class k tasks have similar g/f, we need not show these ratios.) Notice that forming the float time column is analogous to forming a forward difference table. In each of the following examples we assume that Figure 2 is the initial state of priority class k.

(i) Suppose the information vector (with g/f omitted) for T_i is $(T_i, 6, 1, 5)$. Beginning with T_5, we observe that $d_2<d_i\leq d_3<d_4<d_5$. So we insert T_i between T_2 and T_3 and renumber the tasks accordingly. Now $s_2+\tau_2\leq s_3\leq s_4-\tau_3$ since $2+2\leq 5\leq 7-1$, so the priority assignment is complete and all tasks are guaranteed to meet their deadlines. The resulting state of priority class k is shown in Figure 3.

(ii) Suppose instead that the information vector for T_i is $(T_i, 9, 2, 7)$. We find that $d_2<d_i\leq d_3<d_4<d_5$, so we insert T_i between T_2 and T_3 and renumber the tasks accordingly. However, $s_3+\tau_3>s_4$ since $7+2>7$ and a deadline could be missed. But $F_3\geq s_3+\tau_3-s_4$ since $4\geq 7+2-7=2$, so we assign a new start time to T_3: $s_3=s_4-\tau_3=7-2=5$. Now $s_2\leq s_3-\tau_2$ since $2\leq 5-2$, so the priority assignment is complete and all tasks are guaranteed to meet their deadlines. The resulting state of priority class k is shown in Figure 4. (Note the effect of the last in first out rule for breaking ties on start times.)

(iii) Next suppose the information vector for T_i is $(T_i, 9, 4, 5)$. We find that $d_2<d_i\leq d_3<d_4<d_5$, so we insert T_i between T_2 and T_3 and renumber the tasks accordingly. However, $s_3+\tau_3>s_4$ since $5+4>7$, and a deadline could be missed. But $F_3\geq s_3+\tau_3-s_4$ since $4\geq 4$ so we assign a new

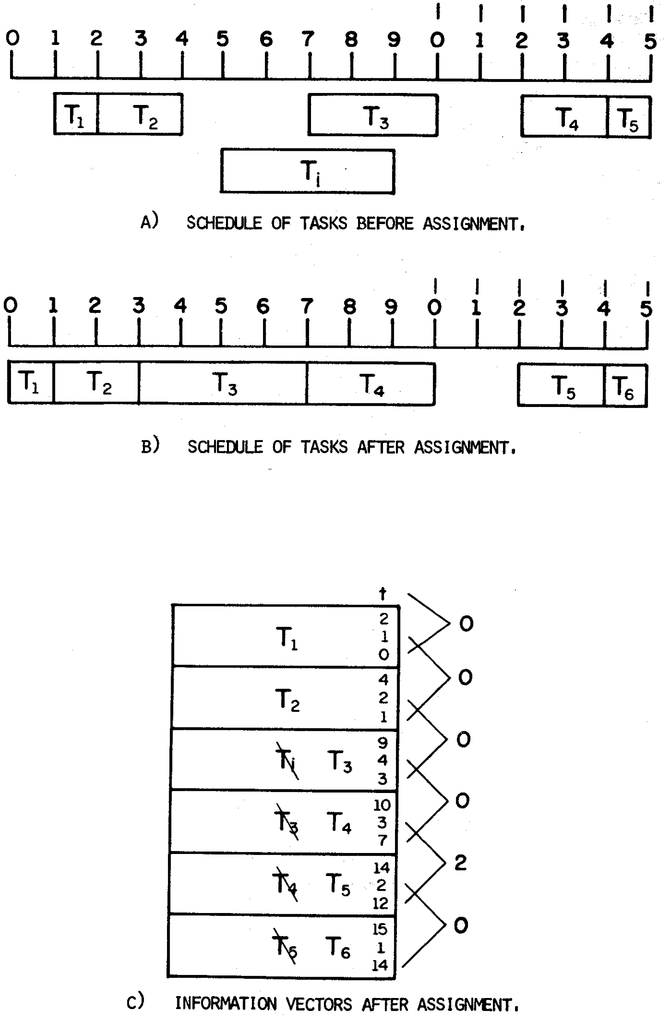
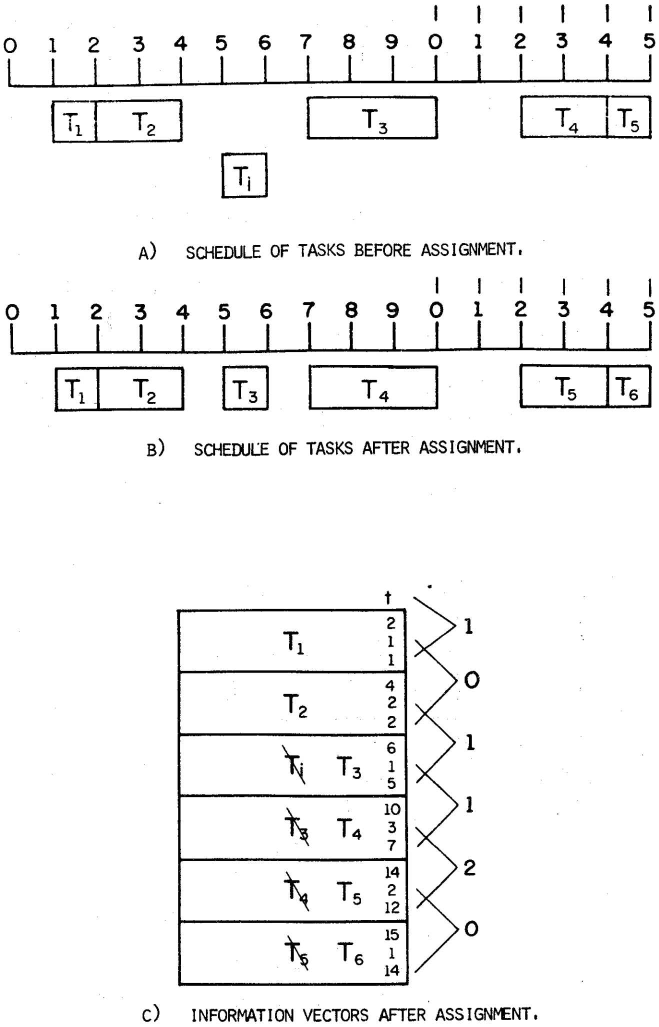


Figure 4—Results of priority assignments for Example (ii)

start time to T_3: $s_3 = s_4 - r_3 = 7 - 4 = 3$. Next $s_2 + r_2 > s_3$ since $2+2>3$, so we assign a new start time to T_2: $s_2 = s_3 - r_2 = 3 - 2 = 1$. Now $s_1 + r_1 > s_2$ since $1+1>1$, so we assign a new start time to T_1: $s_1 = s_2 - r_1 = 0$. The priority assignment is complete and all tasks are guaranteed to meet their deadlines. The resulting state of priority class k is shown in Figure 5.

(iv) As a final example, suppose that the information vector for T_i is $(T_i, 9, 5, 4)$. As before, we find that $d_2 < d_i \leq d_3 < d_4 < d_5$, so we insert T_i between T_2 and T_3 and renumber the tasks accordingly. However, $s_3 + r_3 > s_4$ since $4+5>7$, and a deadline could be missed. Furthermore, $F_3 < s_3 + r_3 - s_4$ since $1 < 4+5-7 = 2$, and the compacting scheme will not help us. Instead we leave the start times at their latest critical values and hope that sufficient float time is created later to enable the tasks to meet their deadlines. The results of this assignment are shown in Figure 6. Note that T_4 is the task which is in danger of missing its deadline.

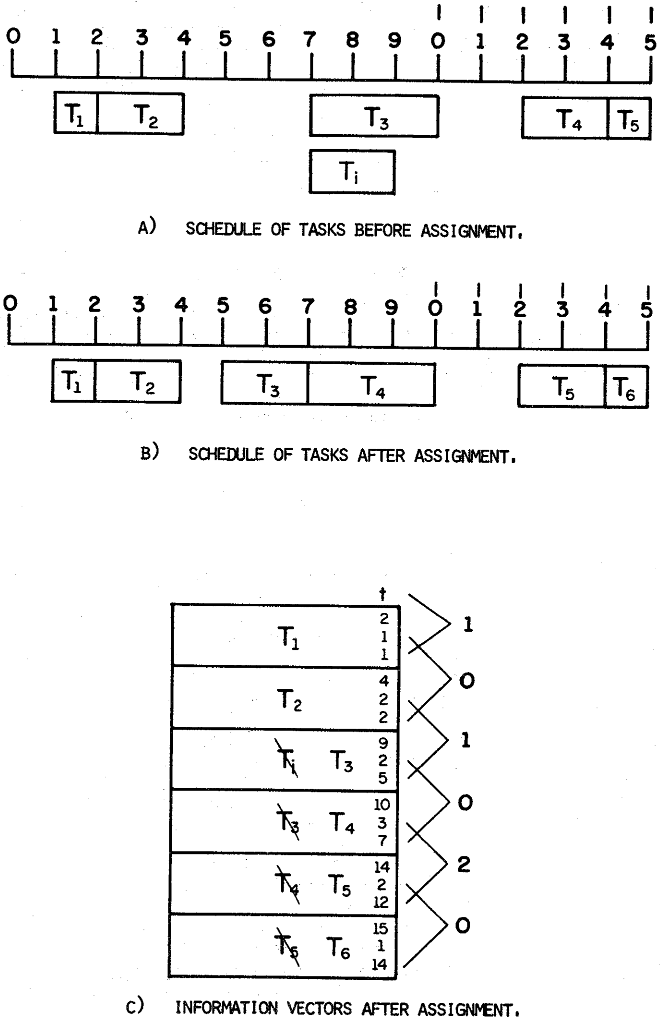


A) SCHEDULE OF TASKS BEFORE ASSIGNMENT.

B) SCHEDULE OF TASKS AFTER ASSIGNMENT.

C) INFORMATION VECTORS AFTER ASSIGNMENT.

Figure 6—Results of priority assignments for Example (iv)



A) SCHEDULE OF TASKS BEFORE ASSIGNMENT.

B) SCHEDULE OF TASKS AFTER ASSIGNMENT.

C) INFORMATION VECTORS AFTER ASSIGNMENT.

Figure 5—Results of priority assignments for Example (iii)

The last example brings up the problem of what to do with a task whose deadline is missed. We simply treat it as though it had just entered the system using the next specified deadline as the current deadline. If no further deadlines are specified, the task is assigned priority $R-1$ and will be processed accordingly.

When a processor finishes executing a task the following scheduling algorithm is used to determine which task is to be processed next. Generally, the algorithm takes the highest priority task in the queue that is closest to its latest starting time.

Scheduling algorithm

Beginning with $k=0$ and using l as an index,

1. We examine the float time, f_1, for the first task in priority class k. Then for $l=k+1$:
2. If f_1 of priority class $k \leq \tau_1$ for the first task of priority class l, we set $k=l$ and continue with Step 1. Otherwise we continue with Step 3.
3. Set $l=l+1$ and continue with Step 2 until all priority classes have been considered. Then continue with Step 4.
4. Assign the first task, T_1, in priority k to the available processor.

The effect of this scheduling algorithm is quite simple. It instructs the scheduler to schedule the important tasks first and then, if there is sufficient time, schedule those lower priority tasks in such a way that as many deadlines as possible are met.

In the foregoing we have tacitly assumed that each task enters the system sufficiently before its deadline to allow processing. The two algorithms taken together facilitate meeting the deadlines, where possible, of the higher priority tasks. Those tasks which do not meet their deadlines will tend to be uniformly late.

LOAD LEVELING IN A NETWORK OF CENTERS

Thus far we have been concerned only with cost effectiveness in a single center. Next, let us consider the more general problem of load leveling within a network of centers. Each center may contain a single computer or a subnet of computers. The topological and physical properties of such networks have been illustrated[4–8] and will not be discussed here.

We wish to determine a strategy which optimizes the value of work performed by the network computer. That is, to guarantee that every task in each center will be processed, if possible, before its deadline and only those tasks that offer the least reward to the network will miss their deadlines. Implicit in this discussion is the simplifying assumption that any task can be performed in any center. This assumption is not as restrictive as it may sound since we can, for the purposes of load leveling, partition a nonhomogeneous network into sets of homogeneous subnetworks which can be considered independently. Thus, in the discussion which follows, we will assume that the network computer is homogeneous.

We define the measure of cost effectiveness for a network of N centers, γ_N, as follows:

$$\gamma_N = \sum_{i=1}^{N} \omega_i \gamma_i \quad \text{given that} \quad \sum_{i=1}^{N} \omega_i = 1 \tag{4}$$

where

> the ω_i are weighting factors that reflect the relative contribution of the i^{th} center to the overall computational capability of the network,

and γ_i is the measure of cost effectiveness for the i^{th} center.

Note that if a center is a subnet of computers, we could employ this definition to determine the measure of cost effectiveness for the subnet. We also let c_{ij} denote the cost of communication between centers i and j; and t_{ij} the transmission time between centers i and j.

Ideally, we want the network computer to operate so that all tasks within the network are processed before their deadlines. If a task is in danger of missing its deadline, we want to consider it as a candidate for transmission to another center for processing. The determination of which tasks should be transferred follows the priority assignment (i.e., priority 0 tasks in danger of missing deadlines should be the first to be considered, priority 1 tasks next, etc.).

We note that this scheme may not discover all tasks that are in danger of missing their deadlines. In order to discover all tasks that might be in danger of missing their deadlines, we would require a look ahead scheme to determine the available float time and to fit lower priority tasks into this float time. The value of such a scheme is questionable, however, since we assume some float time is created during processing and additional float time may be created by sending high priority tasks to other centers. Also, the overhead associated with executing the look ahead scheme would further reduce the probable gain of such a scheme.

The determination of which center should be the recipient of a transmitted task can be determined from the measure of cost effectiveness of each center. Recall that the measure indicates the worth of the work to be processed within a center. Thus a center with a task in danger of missing its deadline will generally have a larger measure than a center with available float time. Thus, by examining the measures for each center, we can determine the likely recipient of tasks to be transmitted. These centers can in turn, examine their own queues and bid for additional work on the basis of their available float times. This approach has a decided economic advantage over broadcasting the availability of work throughout the network and transmitting the tasks to the first center to respond. The latter approach has been investigated by Farber[9] and discarded in favor of bidding.

Once a recipient center has been determined, we would transmit a given task only if the loss in reward associated with not meeting its deadline is greater than c_{ij}, the cost of transmitting the task between centers and transmitting back the results.

When a task is transmitted to a new center its deadline is diminished by t_{ij}, the time to transmit back the results, thus ensuring the task will reach its destination before its true deadline. Similarly, the reward associated with meeting the task's deadline is diminished by c_{ij}, since this represents a reduction in profit. Then the task's g/f ratio is used to determine a new priority and the task is treated like one originating in that center.

This heuristic algorithm provides the desired results that within each center all deadlines are met, if possible, and if any task is in danger of missing its deadline, it is considered for possible transmission to another center which can meet the deadline.

SUMMARY

We have introduced a priority assignment technique which, together with the scheduling algorithm, provides a new approach to resource allocation. The most important innovation in this approach is that it allows a computing installation to maximize reward for the use of resources while allowing the user to specify deadlines for his results. The demand by users upon the resources of a computing installation is translated into rewards for the center.

This approach offers advantages to the user and to the computing installation. The user can exercise control over the processing of his task by specifying its reward/cost ratio which, in turn, determines the importance the installation attaches to his requests. The increased flexibility to the user in specifying rewards for meeting deadlines yields increased reward to the center. Thus the computing installation becomes cost effective, since for a given interval of time, the installation can process those tasks which return the maximum reward. A notable point here, is that this system readily lends itself to measurement.

The measure of cost effectiveness is designed to reflect the status of a center using the priority assignment technique. From its definition, the value of the measure depends not only on the presence of tasks in the system but upon the priority of these tasks. Thus the measure reflects the worth of the tasks awaiting execution rather than just the number of tasks. Therefore, the measure can be used both, statistically, to record the operation of a center, and dynamically, to determine the probability of available float time. This attribute enables us to predict the worth of the work to be performed in any center in the network and facilitates load-leveling between centers.

We have spent a good deal of time discussing what this system does and the problems it attempts to solve. In the interest of fair play, we now consider the things it does not do and the problems it does not solve.

One of the proposed benefits of a network computer is that it is possible to provide, well in advance, a guarantee that, at a predetermined price, a given deadline will be met. This guarantee is especially important for scheduled production runs, such as payroll runs, which must be processed within specified time periods. The system as presented does not directly allow for such a long range guarantee. However, to implement such an option, we simply modify the reward to include the loss of goodwill which would be incurred should such a deadline be missed. Perhaps the easiest way to implement this would be to reserve priority class zero for tasks whose deadlines were previously guaranteed. Under this system we could assure the user, with confidence, that the deadlines could be met at a predetermined (and presumably more expensive) price.

A second problem with the system is that the algorithms do not optimize the mix of tasks which would be processed concurrently in a multiprogramming environment. A common strategy in obtaining a good mix is to choose tasks in such a way that most of the tasks being processed at one time are input/output bound (this is especially common in large systems which can support a large number—five or more—tasks concurrently). Generally smaller tasks are the ones selected to fill this bill. Under our system, the higher priority classes will tend to contain the smaller and less expensive tasks since priority is assigned on the basis of a cost multiplier which is user supplied. We assume a user would be much more reluctant to double (give a reward/cost ratio of 2) the cost of a $100 task than to double the cost of a $5 task. This reluctance will tend to keep a good mix present in a multiprogramming environment.

The final problem we would like to consider is what to do with a task if (horror of horrors) all of its deadlines are missed. There are basically two options, both feasible for certain situations, which will be discussed. The ultimate decision as to which is best rests with the computer center managers. Therefore, we will present the alternatives objectively without any intent to influence that decision.

The first alternative made obvious by the presentation is that when a task misses all of its deadlines the results it would produce are of no further use. Continued attempts to process a task in this instance would be analogous to slamming on the brakes after your car hits a brick wall; simply a waste of resources. Thus, if the deadlines are firm, a center manager could say that a task which misses all of its deadlines should be considered lost to the system.

On the other hand, the results produced by a task could be of value even after the last deadline is missed. In this case the center manager could offer a "low cost" service under which tasks are processed at a reduced rate but at the system's leisure. The danger in this approach is that if run without outside supervision, the system could become saturated with low cost tasks to the detriment of more immediately valuable work. This actually happened at the University of Illinois during early attempts to institute a low cost option. The confusion and headaches which resulted from the saturation were more than enough to justify instituting protective measures. From the results of this experience, it is safe to say that no installation manager will let it happen more than once.

Even in the presence of a few limitations, our system represents a definite positive step in the analysis of network computers. Our approach treats a network computer as the economic entity that it should be: a market place in which vendors compete for customers and in which users contend for scarce resources. The development of this approach is a first step in the long road to achieving economic viability in network computers.

ACKNOWLEDGMENTS

We are particularly grateful to Mr. Einar Stefferud of Einar Stefferud and Associates, Santa Monica, California for his constructive criticism and constant encouragement in this effort. We are also indebted to Professor David J. Farber of the University of California at Irvine, California for many interesting conversations about bidding in distributed networks. Finally, we would like to thank the referees for their careful reviews and suggestions to improve this paper.

This research was supported in part by the National Science Foundation under Grant No. NSF GJ 28289.

REFERENCES

1 E STEFFERUD
Management's role in networking
Datamation Vol 18 No 4 1972

2 J T HOOTMAN
The computer network as a marketplace
Datamation Vol 18 No 4 1972

3 E K BOWDON SR W J BARR
Throughput optimization in network computers
Proceedings of the Fifth International Conference on System Sciences Honolulu 1972

4 N ABRAMSON
The ALOHA system
University of Hawaii Technical Report January 1972

5 H FRANK I T FRISCH
Communication transmission and transportation networks
Addison-Wesley Reading Massachusetts 1971

6 L KLEINROCK
Communication nets stochastic flow and delay
McGraw-Hill New York New York 1964

7 R SYSKI
Introduction to congestion theory in telephone systems
Oliver and Boyd Edinburgh 1960

8 E BOWDON SR
Dispatching in network computers
Proceedings of the Symposium on Computer Communications Networks and Teletraffic April 1972

9 D J FARBER K C LARSON
The structure of a distributed computing system—software
Proceedings of the Symposium on Computer Communications Networks and Teletraffic April 1972

C.mmp—A multi-mini-processor*

by WILLIAM A. WULF and C. G. BELL

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION AND MOTIVATION

In the Summer of 1971 a project was initiated at CMU to design the hardware and software for a multiprocessor computer system using minicomputer processors (i.e., PDP-11's). This paper briefly describes an overview (only) of the goals, design, and status of this hardware/software complex, and indicates some of the research problems raised and analytic problems solved in the course of its construction.

Earlier in 1971 a study was performed to examine the feasibility of a very large multiprocessor computer for artificial intelligence research. This work, reported in the proceedings paper by Bell and Freeman, had an influence on the hardware structure. In some sense, this work can be thought of as a feasibility study for larger multiprocessor systems. Thus, the reader might look at the Bell and Freeman paper for general overview and potential, while this paper has more specific details regarding implementation since it occurs later and is concerned with an active project. It is recommended that the two papers be read in sequence.

The following section contains requirements and background information. The next section describes the hardware structure. This section includes the analysis of important problem in the hardware design: interference due to multiple processors accessing a common memory. The operating system philosophy, and its structure is given together with a detailed analysis of one of the problems incurred in the design. One problem is determining the optimum number of "locks" which are in the scheduling primitives. The final section discusses a few programming problems which may arise because of the possibilities of parallel processing.

REQUIREMENTS

The CMU multiprocessor project is designed to satisfy two requirements:

1. particular computation requirements of existing research projects; and
2. research interest in computer structures.

The design may be viewed as attempting to satsify the computational needs with a system that is conservative enough to ensure successful construction within a two year period while first satisfying this constraint, the system is to be a research vehicle for multiprocessor systems with the ability to support a wide range of investigations in computer design and systems programming.

The range of computer science research at CMU (i.e., artificial intelligence, system programming, and computer structures) constrains processing power, data rates, and memory requirements, etc.

(1) The artificial intelligence research at CMU concerned with speech and vision imposes two kinds of requirements. The first, common to speech and vision, is that special high data rate, real time interfaces are required to acquire data from the external environment. The second more stringent requirement, is real time processing for the speech-understanding system. The forms of parallel computation and intercommunication in multiprocessor is a matter for intensive investigation, but seems to be a fruitful approach to achieve the necessary processing capability.

(2) There is also a significant effort in research on operating systems and on understanding how software systems are to be constructed. Research in these areas has a strong empirical and experimental component, requiring the design and construction of many systems. The primary

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-0107) and is monitored by the Air Force Office of Scientific Research.

requirement of these systems is isolation, so they can be used in a completely idiosyncratic way and be restructured in terms of software from the basic machine. These systems also require access by multiple users and varying amounts of secondary memory.

(3) There is also research interest in using Register Transfer Modules (RTM's) developed here and at Digital Equipment Corporation (Bell, Grason, et al., 1972) and in production as the PDP-16 are designed to assist in the fabrication of hardware/software systems. A dedicated facility is needed for the design and testing of experimental system constructed of these modules.

TIMELINESS OF MULTIPROCESSOR

We believe that to assemble a multiprocessor system today requires research on multiprocessors. Multiprocessor systems (other than dual processor structures) have not become current art. Possibly reasons for this state of affairs are:

1. The absolutely high cost of processors and primary memories. A complex multiprocessor system was simply beyond the computational realm of all but a few extraordinary users, independent of the advantage.
2. The relatively high cost of processors in the total system. An additional processor did not improve the performance/cost ratio.
3. The unreliability and performance degradation of operating system software,—providing a still more complex system structure—would be futile.
4. The inability of technology to permit construction of the central switches required for such structures due to low component density and high cost.
5. The loss of performance in multiprocessors due to memory access conflicts and switching delays.
6. The unknown problems of dividing tasks into subtasks to be executed in parallel.
7. The problems of constructing programs for execution in a parallel environment. The possibility of parallel execution demands mechanisms for controlling that parallelism and for handling increased programming complexity.

In summary, the expense was prohibitive, even for discovering what advantages of organization might overcome any inherent decrements of performance. However, we appear to have now entered a technological domain when many of the difficulties listed above no longer hold so strongly:

1′. Providing we limit ourselves to multiprocessors of minicomputers, the total system cost of processors and primary memories are now within the price range of a research and user facility.

2′. The processor is a smaller part of the total system cost.

3′. Software reliability is now somewhat improved, primarily because a large number of operating systems have been constructed.

4′. Current medium and large scale integrated circuit technology enables the construction of switches that do not have the large losses of the older distributed decentralized switches (i.e., busses).

5′. Memory conflict is not high for the right balance of processors, memories and switching system.

6′. There has been work on the problem of task parallelism, centered around the ILLIAC IV and the CDC STAR. Other work on modular programming [Krutar, 1971; Wulf, 1971] suggests how subtasks can be executed in a pipeline.

7′. Mechanisms for controlling parallel execution, *fork-join* (Conway, 1963), P and V (Dijkstra, 1968), have been extensively discussed in the literature. Methodologies for constructing large complex programs are emerging (Dijkstra, 1969, Parnas, 1971).

In short, the price of experimentation appears reasonable, given that there are requirements that appear to be satisfied in a sufficiently direct and obvious way by a proposed multiprocessor structure. Moreover, there is a reasonable research base for the use of such structures.

RESEARCH AREAS

The above state does not settle many issues about multiprocessors, nor make its development routine. The main areas of research are:

1. The multiprocessor hardware design which we call the PMS structure (see Bell and Newell, 1971). Few multiprocessors have been built, thus each one represents an important point in design space.
2. The processor-memory interconnection (i.e., the switch design) especially with respect to reliability.

3. The configuration of computations on the multiprocessor. There are many processing structures and little is known about when they are appropriate and how to exploit them, especially when not treated in the abstract but in the context of an actual processing system:

   Parallel processing: a task is broken into a number of subtasks and assigned to separate processors.

   Pipeline processing: various independent stages of the task are executed in parallel (e.g., as in a co-routine structure).

   Network processing: the computers operate quasi-independently with intercommunication (with various data rates and delay times).

   Functional specialization: the processors have either special capabilities or access to special devices; the tasks must be shunted to processors as in a job shop.

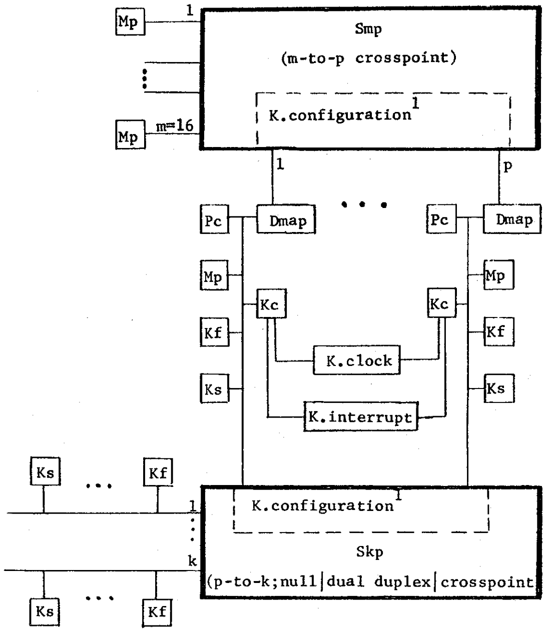
   Multiprogramming: a task is only executed by a single processor at a given time.

   Independent processing: a configurational separation is achieved for varying amounts of time, such that interaction is not possible and thus doesn't have to be processed.
4. The decomposition of tasks for appropriate computation. Detailed analysis and restructuring of the algorithm appear to be required. The speech-understanding system is one major example which will be studied. It is interesting both from the multiprocessor and the speech recognition viewpoints.
5. The operating system design and performance. The basic operating system design must be conservative, since it will run as a computation facility, however it has substantial research interest.
6. The measurement and analysis of performance of the total system.
7. The achievement of reliable computation by organizational schemes at higher levels, such as redundant computation.

THE HARDWARE STRUCTURE

This section will briefly describe the hardware design without explicitly relating each part to the design constraints. The configuration is a conventional multiprocessor system. The structure is given in Figure 1.

There are two switches, Smp and Skp, each of which provide intercommunication among two sets of components. Smp allows each processor to communicate with all primary memories (in this case core). Skp allows each processor (Pc), to communicate with the various controllers (K), which in turn manage the secondary memories (Ms), and I/O devices transducers (T). These switches are under both processor and manual control.



where: Pc/central processor; Mp/primary memory; T/terminals;
Ks/slow device control (e.g., for Teletype);
Kf/fast device control (e.g., for disk);
Kc/control for clock, timer, interprocessor communication

[1] Both switches have static configuration control by manual and program control

Figure 1—Proposed CMU multiminiprocessor computer/C.mmp

Each processor system is actually a complete computer with its own local primary memory and controllers for secondary memories and devices. Each processor has a Data operations component, Dmap, for translating addresses at the processor into physical memory addresses. The local memory serves both to reduce the bandwidth requirements to the central memory and to allow completely independent operation and off-line maintenance. Some of the specific components shown in Figure 1 are:

K.clock: A central clock, K.clock, allows precise time to be measured. A central time base is broadcast to all processors for local interval timing.

K.interrupt: Any processor is allowed to generate an interrupt to any subset of the Pc configuration at any of several priority levels. Any processor may also cause any subset of the configuration to be stopped and/or restarted. The ability of a processor to interrupt, stop, or restart another is under both program and manual control. Thus, the console loading function is carried out via this mechanism.

Smp: This switch handles information transfers between primary memory processors and I/O devices. The switch has ports (i.e., connections) for m busses for primary memories and p busses for processors. Up to min(m,p) simultaneous conversations possible via the cross-point arrangement.

Smp can be set under programmed control or via manual switches on an override basis to provide different configurations. The control of Smp can be by any of the processors, but one processor is assigned the control.

Mp: The shared primary memory, Mp, consists of (up to) 16 modules, each of (up to) 65k, 16 bit, words. The initial memories being used have the following relevant parameters: core technology; each module is 8-way interleaved; access time is 250 nanoseconds; and cycle time is 650 nanoseconds. An analysis of the performance of these memories within the C.map configuration is given in more detail below.

Skp: Skp allows one or more of k Unibusses (the common bus for memory and i/o on an isolated PDP-11 system) which have several slow, Ks (e.g., teletypes, card readers), or fast controllers, Kf, (e.g., disk, magnetic tape), to be connected to one of p central processors. The k Unibusses for the controllers are connected to the p processor Unibusses on a relatively long term basis (e.g., fraction of a second to hours). The main reasons for only allowing a long term, but switchable, connection between the k Unibusses and the processor is to avoid the problem of having to decide dynamically which of the p processors manage a particular control. Like Smp, Skp may be controlled either by program or manually.

Pc: The processing elements, Pc, are slightly modified versions of the DEC PDP-11. (Any of the PDP-11 models may be intermixed.)

Dmap: The Dmap is a Data operations component which takes the addresses generated in the processor and converts them to addresses to use on the Memory and Unibusses emanating from the Dmap. There are four sets of eight registers in Dmap, enabling each of eight 4,096 word blocks to be relocated in the large physical memory. The size of the physical Mp is 2^{20} words (2^{21} bytes). Two bits in the processor, together with the address type are used to specify which of the four sets of mapping registers is to be used.

Dmap

The structure of the address map, is described below and in Figure 2 together with its implications for two kinds of programs: the user and the monitor programs. For the user program, the conventional PDP-11 addressing structure is retained—except that a program does not have access to the "i/o page," and hence the full 16-bit address space refers to the shared primary memory.

A PDP-11 program generates a 16-bit address, even though the Unibus has 18-bit addressing capability. In this scheme the additional two address bits are obtained from two unused program status (PS) register bits. (Note, this register is inaccessible to user pro-

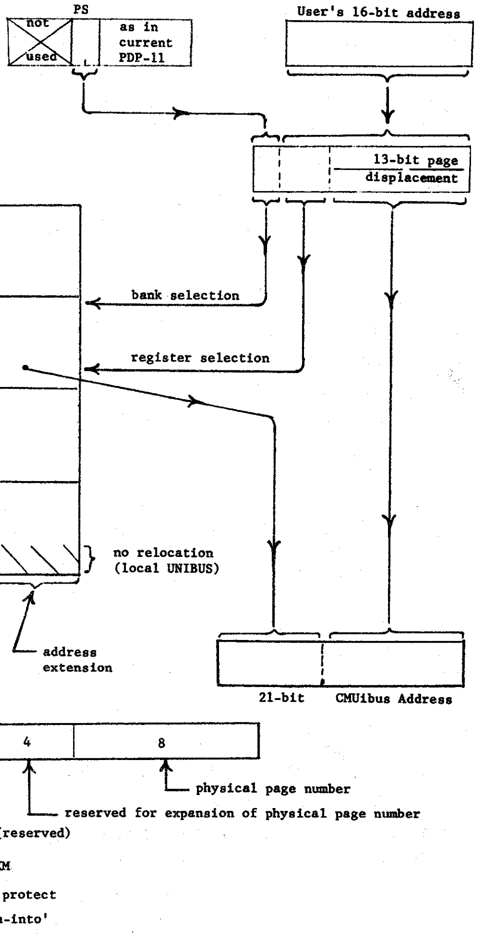


Figure 2—Format of data in the relocation registers

grams.) These are two additional bits, provides four addressing modes:

00-mode
01-mode
10-mode } These addresses are always mapped, and always refer to the shared, large, primary memory.

11-mode All but 8 kw (kilo words) of this address space is mapped as above. The 8 kw of this space which is not mapped refers to the private Unibus of each processor; 4 kw of this space is for private (local) memory and 4 kw is used to access i/o devices attached to the processor.

For mapped references, the mapping consists of using the most significant five bits of the 18-bit address to select one of 30 relocation registers, and replacing these by the contents of the 8 low order bits of that register yielding an overall 21-bit address. Alternatively, consider that two bits of the PS select one of four banks of relocation registers and the leftmost three bits of the users (16-bit) address select one of the eight registers in this bank (six in bank three). A program may (by appropriate monitor calls) alter the contents of the relocation registers within that bank and thus alter its "instantaneous virtual memory"—that is, the set of directly addressable pages. The format of each of the 30 relocation registers is as also shown in Figure 2 where:

1. The 'written-into' bit is set (to 1) by the hardware whenever a write operation is performed on the specified page.
2. The 'write protect' bit, when set, will cause a trap on (before) an attempted write operation into the specified page.
3. The NXM, 'non-existent memory', when set, will cause a trap on any attempted access to the specified page. Note: this is not adequate for, nor intended for, 'page fault' interruption.
4. The 8-bit 'physical page number' is the actual relocation value.

THE MEMORY INTERFERENCE PROBLEM

One of the most crucial problems in the design of this multiprocessor is that of the conflict of processor requests for access to the shared memories.

Strecker (1970) gives closed form solutions for the interference in terms of a defined quantity, the UER (unit execution rate). The UER is, effectively, the rate memory references and, for the PDP-11, is approximately twice the actual instruction execution rate. (Although a single instruction may make from one to five memory references, about two is the average.) Neglecting i/o transfers*, assuming access requests to memories at random, and using the following mean parameters:

t_p the time between the completion of one memory request and the next request

t_a, t_c the access time and cycle time for the memories to be used

$t_w = t_c - t_a$ the rewrite time of the memory

Strecker gives the following relations:

$$t_p = t_w: \quad \text{UER} = (m/t_c)\,(1 - (1 = 1/m)^p)$$

$$t_p < t_w: \quad \text{UER} = \frac{m}{t} \times \frac{1 - (1 - 1/m)^p}{1 - (1 - 1/m)^p}$$

$$t_p > t_w: \quad \text{UER} = (m/t_c)(1 - (1 - P_m/m)^p)$$

$$\text{where } P_m + (m/p)(\frac{t_p - t_r}{t_c})(1 - (1 - P_m/m^p)) - 1 = 0$$

Various speed processors, various types of memories, and various switch delays, t_d, can be studied by means of these formulas. Switch delays effects are calculated by adding to t_a and t_c, i.e., $t_a' = t_d + t_a$; and $t_c' = t_d + t_c$. For example, the following cases are given in the attached graphs. The graphs show UER × 10^6 as a function of p for various parameters of the memories. The two values of t_d shown correspond to the estimated switch delay in two cable-length cases: 10′ and 20′. The t_c, t_a values correspond to six memory systems which were considered. The value of t_p is that for the PDP-11 model 20.

Given data of the form in Figures 3 and 4 it is possible to obtain the cost effectiveness of various processor-memory configurations. An example of this information for a particular memory configuration (16 memories, t_c = 400) and three different processors (roughly corresponding to three models of the PDP-11 family) is plotted in Figure 5. Note that a small configuration of five Pc.1's has a performance of 4.5 × 10^6 accesses/second (UER). The cost of such a system is approximately $375K, yielding a cost-effectiveness of 12. Replacing these five processors with the same number of Pc.3's yields a UER of 15 × 10^6 for about $625K, or a cost-effectiveness of about 24. Following this strategy provides a very cost-effective system once a reasonably large number of processors are used.

* A simple argument indicates that i/o traffic is relatively insignificant, and so has not been considered in these figures. For example, transferring with four drums or 15 fixed head disks at full rate is comparable to one Pc.

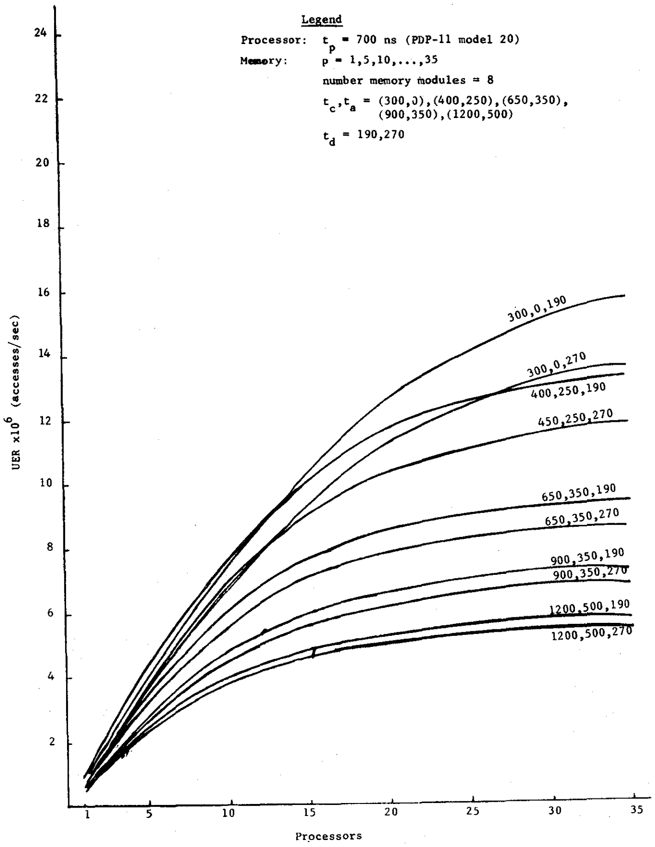


Figure 3—Performance for various memory-processor configurations

In fact, in the range 15–30 processors the cost-effectiveness is relatively constant while the absolute performance nearly doubles.

Unfortunately these studies of memory interference assume a random distribution of memory references—an assumption may be invalid when true parallel processing is performed (notably if shared programs are executed, as in the operating system). Several approaches to predicting and preventing these conflicts are being studied:

Software page-placements

Better-than-random reference patterns may be achieved by having the operating system page-placement algorithms attempt to localize process' pages within a single memory module. No results on this approach have been obtained to date.

Switch, Smp, measurement

Schemes for dynamically measuring the Mp-Pc reference pattern are being considered. The most accurate method under consideration is to associate a small memory with each crosspoint intersection. This can be constructed efficiently by having a memory array for each of the m rows, since control is on a row (per memory) basis. When each request for a particular row is acknowledged, a 1 is added to the register corresponding to the procesor which gets the request. These data could then serve as input to algorithms of the type described under (1). Such a scheme has the drawback of adding hardware (cost) to the switch, and possibly lowering reliability. Since the performance measures given earlier are quite good, even for large numbers of processors, this approach does not seem justified at this time.

A cache

Since performance for all but shared programs may approximate the random references assumption of Strecker's analysis, special provision for these references might be provided. The addition of a cache memory between Dmap and Smp allows programs to migrate

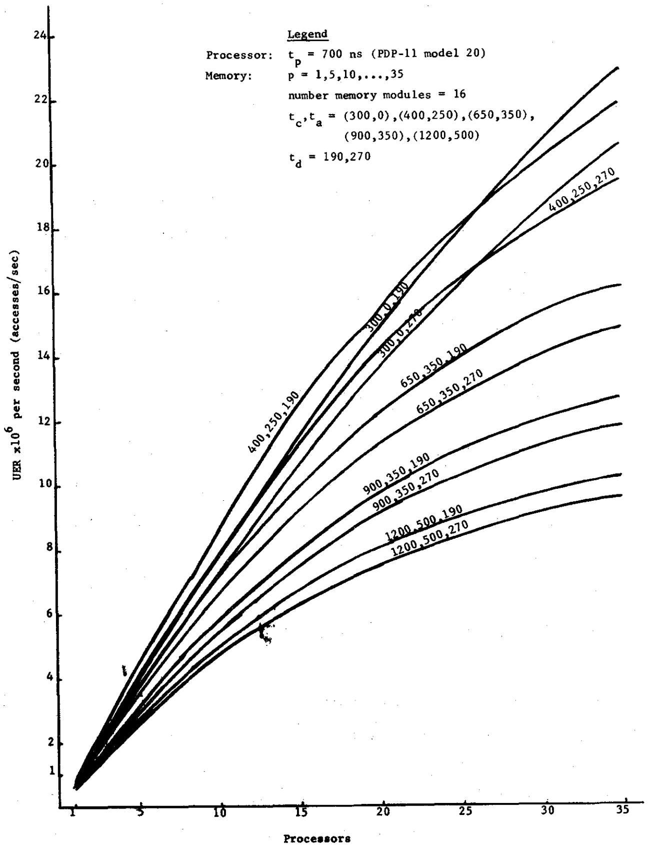


Figure 4—Performance for various memory-processor configurations

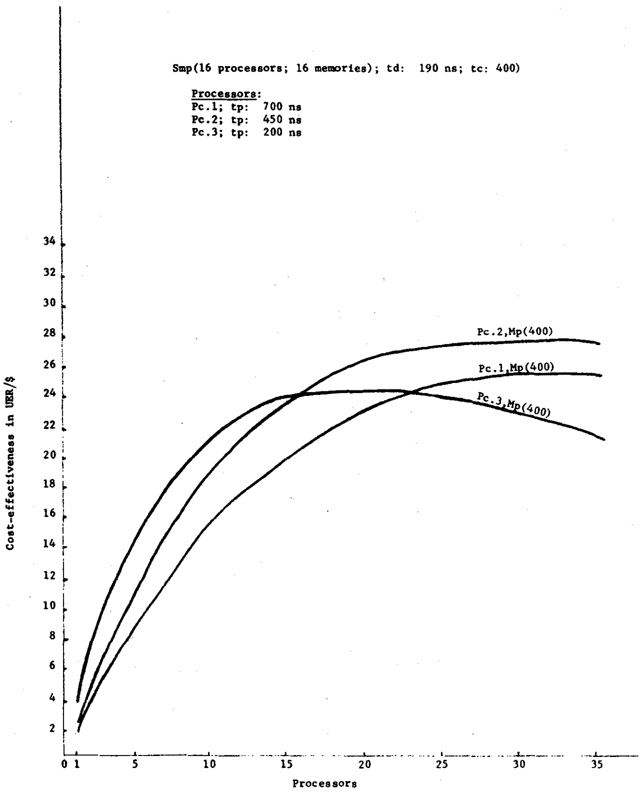


Figure 5—Cost effectiveness (UER/$)

into the cache thereby diminishing the number of requests for a single memory. This also provides faster access since the Smp is avoided.

By introducing such a cache, however, a potential problem is created regarding the validity of data since it might be possible to have sixteen different values of a single variable at a given instant of time. A scheme for avoiding this is to allow only information from "read only" pages (especially instructions) to appear in the cache. (In particular, the bit marked 'reserved' in Figure 2 is used to signal that data from the page may be placed into the cache.) Traces of PDP-11 programs executions indicate that a small cache (256–512 words) will capture 70–90 percent of the eligible references and 40–50 percent of all references. McCredie (1972) has studied the effect of such a cache on overall system performance both analytically and by simulation. The results of these studies indicate an improvement of 10–40 percent in overall system performance.

THE OPERATING SYSTEM

Although the technology of operating systems has made significant progress in the past decade, there are virtually no extant examples of systems constructed specifically for multiprocessor environments. In particular, no systems have been built to support the variety of process relations (parallel, pipeline, etc.) envisioned for C.mmp. Moreover, there is a relative lack of experience in organizing computations for parallel execution. These facts have driven the operating system design to the following, hopefully conservative, position:

> The operating system will consist of a "kernel" and a "standard Extension." The kernel will provide a set of mechanisms (tools) for building an operating system, but no policies (e.g., no scheduler, no file structure, no. . .). The standard extension will implement an (easily modified or replaced) set of "conventional" operating system facilities (e.g., a scheduler, file system, . . .). The kernel will support the (simultaneous) execution of an (almost) arbitrary number of extensions.

Under this strategy the variety of computational structures is not *a priori* limited by the structure of the underlying system. There are also potential hazards in the kernel approach. One of them is the possibility that extension in some (important) desired direction is not possible because of irrevocable decision made too early (though this problem is hardly unique to the kernel approach). Another hazard is that intolerable overhead might accrue by enforced multiple 'layering' of extensions. Both analysis and simulated use indicated that neither of these problems exist for the proposed design.

The remainder of this section is devoted primarily to a description of the kernel (called HYDRA).

In considering what set of mechanisms (tools) should be provided by an operating system kernel two commonly held views of the essential nature of an operating system are relevant:

—An operating system creates a "virtual machine" to support (user) programs by providing resources and operations not present in the underlying hardware (e.g., "files," file "read" and "write" operations, etc.).

—An operating system is a resource (virtual and physical) manager and allocator.

Note the emphasis in both views on resources; their creation, management, and operations on them. From these views we infer than an appropriate set of tools for building an operating system must provide for:

—the creation of new virtual resources;

—the 'representation' of a new resource in terms of existing ones;

—the creation of operations on resources and/or their representation; and

—protection (against illegal operations on a resource), both
  (a) uniformly over a class of resources; and
  (b) with regard to specific instances of a resource.

This list serves as the design goals for HYDRA against which the design is evaluated.

Since the resources are central to the design, we define a suitable abstraction of these called an *object*; objects are the basic entity of interest in HYDRA. An object has a *name*, a *type*, and usually some other (type dependent) information associated with it. The name of every object is unique and is called its *global name*. There is a supply of unique global names to last over the system's total life. Thus, it is not possible for two (or more) objects to have the same global name.

The set of extant objects is partitioned into equivalence classed by their types. There is also an unlimited supply of object types—new types may be created at will. The initial system includes a particular object whose name is TYPE. New types are created by creating an object whose type is TYPE; thus a class of objects of a particular type are "represented" by an object of type TYPE. Suppose, for example, one wished to create a new kind of virtual resource. This would be done by creating an object (assume its name is X) of type TYPE. The object X now serves as a representative for all particular instances of resources of this new variety; in particular, objects of type X may now be created to represent the instances of the new resource.

Operations are performed on objects by procedures.* A procedure is an object of type PROCTYPE. The 'right' to invoke a procedure on each particular object is limited by both the type of object and the user's access to it (see below).

During execution of a procedure there exists a *local name space*, *lns*, associated with it. The *lns* is an object which provides a mapping between local object names (integers) accessible to the procedure and the actual global names for objects. Each *lns* entry may also restrict the access rights (procedures that can be invoked to perform operations on or with the object) to a subset of those defined for that type of object. Thus the *lns* provides both mapping and protection functions.

The only primitive operations in the system which are *not* provided by procedures are CALL and RETURN, whose functions are, respectively, to permit entry to, and exit from, procedures. CALL also provides parameter checking and establishes the *lns* for the called procedure.

To recap: The primitive notions in HYDRA are those of an *object*, a *global name*, and a *type*. Some specific types are TYPE, PROCTYPE, and LNSTYPE. Procedure objects may be invoked by a CALL and are exited by a RETURN. Protection is provided by: (1) restricting access to objects to those named in the current *lns*, (2) restricting the operations (procedures) which may be applied to an object to those associated with that type of object, and (3) further restricting the set of operations which may be applied to any object named in an *lns* to a subset of those in (2).

Figure 6, gives a concrete example of this mechanism. Suppose that a new type of object, a "bibliography file," is created. Three specific operations are permitted on these objects: updating, printing, and erasing. Therefore three procedure objects UPDATE, PRINT, and ERASE are created to perform these

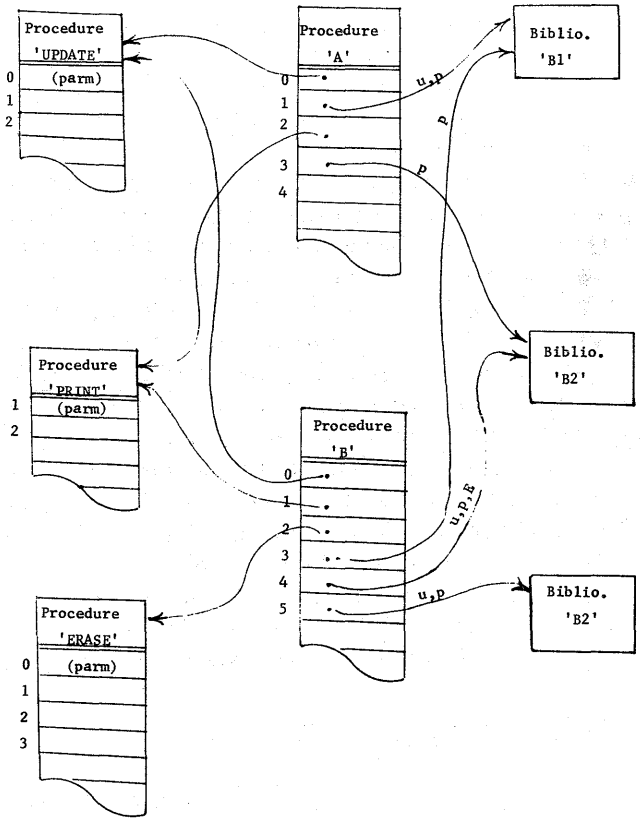


Figure 6—Example of LNS mapping and protection

* Here we wish to invoke the reader's intuitive notion of a 'procedure' and its properties, e.g., a body of code, local storage, a parameter mechanism, etc.

operations; no other operations are permitted on this type of object. The situation in Figure 6 might exist at some instant. It shows (in the center) two procedures, A and B, and their associated lns's—directed arcs indicate the mapping function of the *lns* and the letters along an arc indicate permitted accesses. Here, local name '1' of procedure A references a particular bibliography object, B1; UPDATE and PRINT access by A are permitted. The following information can be observed from the diagram:*

—A.0 → UPDATE
—B.0 → UPDATE
—A.2 → PRINT
(and so on; note that A cannot name the ERASE procedure nor bibliography object B2)
—A may: update and print B1; only print B2
—B may: only print B1; update, print, or erase B2; update and print B3.

THE RELIABILITY PROBLEM

The existence in the physical system of multiple, redundant resources suggests the possibility of highly reliable operation—at least in the sense of continuing to provide (degraded) service when some fraction of the hardware is down. An explicit goal in the HYDRA design is to provide commensurate reliability in the software. Reliability may have two components:

(1) Correctness: The major reason for unreliability in current software is that it is incorrect. However,
—the proposed design for the kernel is small enough that a "constructive programming" approach can be used effectively (Dijkstra)
—the design suggests natural modular decomposition along the lines suggested by Parnas (Parnas 1972)
—the coding is being done in a "systems implementation language" (Bliss/11) (Wulf, et al., 1970, 1971)
—the protection mechanism itself absolutely guarantees that an erroneous or malicious program cannot destroy information to which it does not have legal access.
Therefore the correctness of the kernel must be proven and its construction is proceeding in a highly stylized form design to facilitate this.

(2) Malfunction: Even if the software is correct it is possible for the system to be unreliable, for example, as the result of misexecution of correct code by (perhaps intermittently) failing hardware. This problem is compounded by both the multiprocessor character of the system and the kernel design.

Although a great deal of research has been done on hardware reliability, (for example in connection with computers for extended space missions and electronic telephone switching systems), little has been done on software reliability. Undoubtedly this situation has resulted from the fact correctness (or lack of it) rather than malfunction has been the primary cause of unreliable software.

Possibly some of the ideas from the work on hardware reliability can be carried over to software; a few of these are discussed below. It should be remembered that there is a cost/effectiveness trade-off in each of these—an increasing degree of reliability may be achieved only at an increased cost. A very high degree of reliability appears expensive and probably unnecessary in any case.

Redundancy

One of the common forms of fault detection is to replicate a critical component and, at appropriate points, to verify that the components agree. This might appear in several forms in software:

—Critical computations might be performed by two distinct methods within a single processor and their results compared
—The same code for a critical computation might be performed by two distinct processors and their results compared
—Multiple copies of critical data might be stored on distinct devices and their contents compared.

Consistency

A less demanding (and expensive) form of fault detection is to merely check the reasonableness of a computation or data item value. A simple example is for all lists to be stored in "circular, doubly-linked" form since this permits a check that the predecessor and successor of an item correctly point to the item. Another example of the same kind is for critical items to carry a "self-identification" which is checked before any updates to the item are made.

* The notation X.n will be used to refer to the *n*th local name in procedure *X*; "→" is to be read "maps onto" or "is a reference to."

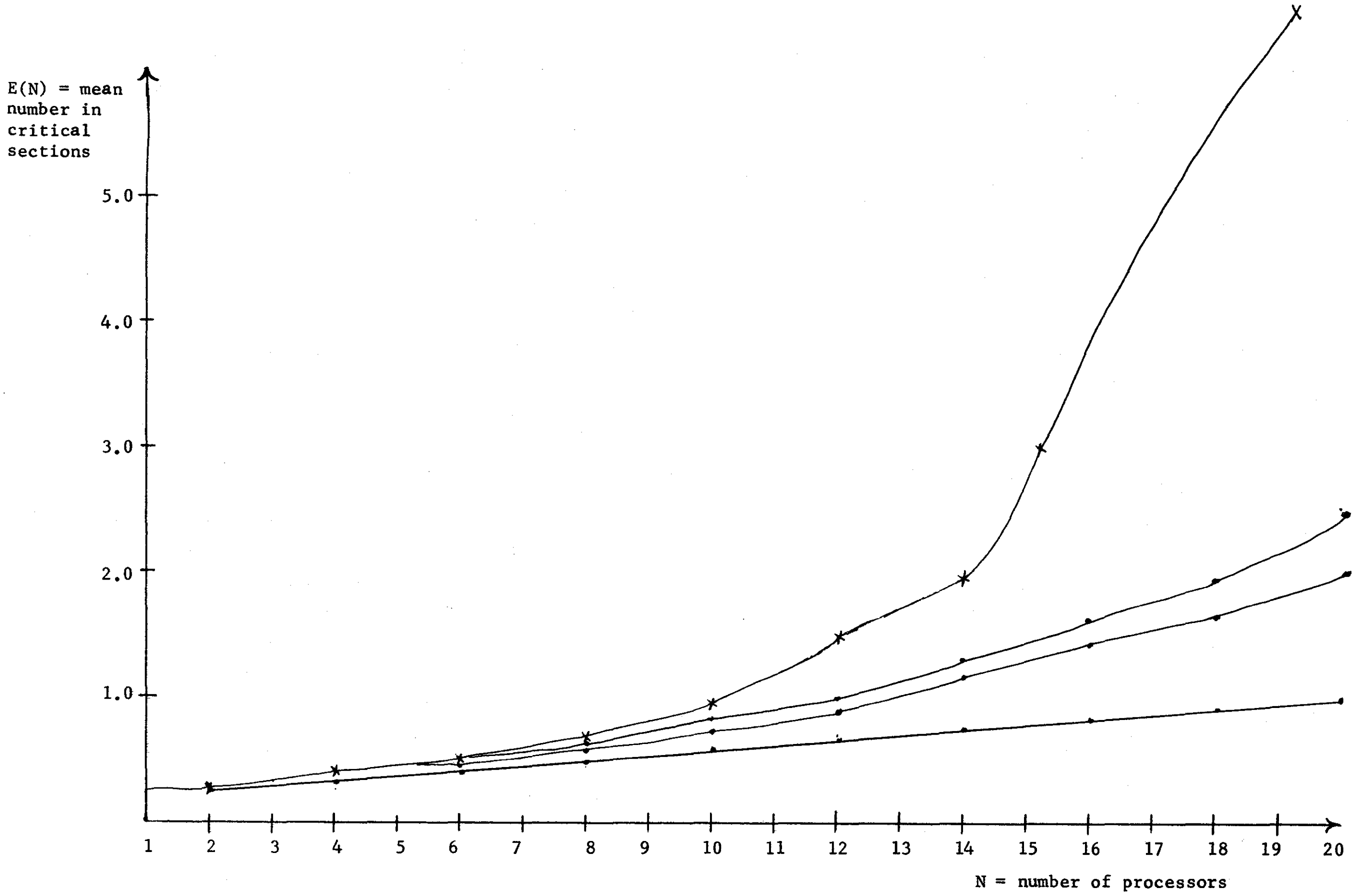


Figure 7—Mean response time for scheduling

Diagnostics

An even less demanding scheme is to attempt to ascertain whether the hardware is functioning properly before faults occur in critical places. This might be done on the fly just before a critical computation is performed, at fixed intervals, or simply whenever the processor is not occupied with other tasks.

THE LOCK PROBLEM

An interesting problem in the design of a multiprocessor operating system is scheduling and coordinating the many, individual processors. In HYDRA the information necessary to make these decisions is represented in a shared data base and the program(s) which make the decision may be executed on any of the processors—and possibly on several processors simultaneously. While one processor is accessing or updating this shared information all other processors must be prevented from accessing and/or changing it. The act of protecting a data item is called "locking" and that portion of a program which accesses a locked item is called a "critical section."

A basic design problem in such a scheduler is to determine the number of critical sections, S, that will maximize system performance. At one extreme a single lock could be used for the entire data base; at the other extreme each item could have a separate lock. Since a finite time, L, is required to perform the locking operations, the overhead due to this operation is minimized for S = 1. However, if S = 1, the possibility of several processors performing scheduling operations simultaneously is precluded. Even though the performance for each individual processor is degraded, total system performance may be improved by choosing S > 1.

A report by McCredie (McCredie, 1972) discusses two analytic models which have been used to study this problem; here we shall merely indicate the results. Figure 7 illustrates the relationship predicted by one of McCredie's models between the mean response time to a scheduling request, the number of critical sections, and the number of processors.

Mean response time increases with the number of processors. For S constant, the increase in mean response time is approximately linear, with respect to N, until the system becomes congested. As N increases beyond this point, the slope grows with increasing N.

The addition of one more critical section significantly improves mean response, for higher values of N, in both models. The additional locking overhead, L, associated with each critical section degrades performance slightly for small values of N. At these low values of N, the rate of requests is so low that the extra locking overhead is not compensated for by the potential parallel utilization of critical sections.

The most interesting characteristic of these models is the large performance improvement achieved by the creation of a small number of additional critical sections. The slight response time degradation for low arrival rates indicates that an efficient design would be the implementation of a few (S=2, 3 or 4) critical sections. This choice would create an effective safety valve. Whenever the load would increase, parallel access to the data would occur and the shared scheduling information would not become a bottleneck. The overhead at low arrival rates is about 5 percent percent and the improvement at higher request rates is approximately 50 percent.

Given the dramatic performance ratios predicted by these modes, the HYDRA scheduler was designed so that S lies in the range 2-7 (the exact value of S depends upon the path through the scheduler).

PROGRAMMING ISSUES

Thus far both highly general and highly specific aspects of the hardware and operating system design of C.mmp have been described. These alone, however, do not provide a complete computing environment in which productive research can be performed. An environment of files, editors, compilers, loaders, debugging aids, etc., must be available. To some extent existing PDP-11 software can and will be used to supply these facilities. However, the special problems and potentials of a multiprocessor preclude this from being a totally appropriate set of facilities.

The potential of true parallel processing obviously requires the introduction of language and system facilities for creating and synchronizing sub-tasks. Various proposals for these mechanisms have existed for some time, such as fork-join, "P" and "V", and they are not especially difficult to add to most existing languages, given the right basic hardware. Parallelism has a more profound effect on the programming environment, however, than the perturbations due to a few language constructs. The primary impact of parallelism is in the increase in complexity of a system due to the possible interactions between its components. The need is not merely for constructs to invoke and control parallel programs, but for conceptual tools dealing with the complexity of programs that can be fabricated with these constructs.

In its role as a substrate for a number of rearch projects, C.mmp has spawned a project to investigate the conceptual tools necessary to deal with complex programs. The premise of this research is that the approach to building large complex programs, and especially those involving parallelism, is essentially methodological in nature: the primitives, i.e., language features, from which a program is built are not nearly as important as the *way* in which it is built. Two particular methodologies—"top-down design" or "structured programming" (Dijkstra, 1969) and "modular decomposition" (Parnas, 1971) have been studied by others and form starting points for this research.

While the solution to building large systems may be methodological, not linguistic, in nature, one can conceive of a programming environment, including a language, whose structure facilitates and encourages the use of such a methodology. Thus the context of the research has been to define such a system as a vehicle for making the methodology explicit. Although they are clearly not independent, the language and system issues can be divided for discussion.

Language issues

Most language development has concerned itself with "convenience"—providing mechanisms through which a programmer may more conveniently express computation. Language design has largely abdicated responsibility for the programs which are synthesized from the mechanisms it provides. Recently, however, language designers have realized that a particular construct, the general *goto*, can be (mis)used to easily synthesize "faulty" programs and a body of literature has developed around the theoretical and practical implications of its removing from programming languages (Wulf, 1971a).

At the present stage of this research it is easier to identify constructs which, in their full generality, can be (mis) used to create faulty programs than to identify forms for the essential features of these constructs which cannot be easily misused. Other constructs are:

Algol-like scope rules

The intent of scope rules in a language is to provide protection. Algol-like scope rules fail to do this in two ways. First, and most obviously, these rules do not distinguish kinds of access; for example, "read-only" access is not distinguished from "read-write" access. Second, there is no natural way to prevent access to a variable at block levels "inside" the one at which it is declared.

Encoding

A common programming practice is to encode information, such as age, address, and place of birth, in the available data types of a language, e.g., integers. This is necessary, but leads to programs which are difficult to modify and debug if the manipulation of these encodings is distributed throughout a large program.

Fixed representations

Most programming languages fix both syntactic and run-time representations; they enforce distinctions between macros and procedures, data and program, etc., and they provide irrevocable representations of data structures, calling sequences, and storage allocation. Fixed representations force programmers to make decisions which might better be deferred and, occasionally, to circumvent the fixed representation (e.g., with in-line code).

SYSTEMS ISSUES

Programming should be viewed as a process, not a timeless act. A language alone is inadequate to support this process. Instead, a total system that supports all aspects of the process is sought. Specifically, some attributes of this system must be:

(a) To retain the constructive path in final and intermediate versions of a program and to make this path serve as a guide to the design, construction, and understanding of the program. For example, the source (possibly in its several representations) corresponding to object code should be recoverable for debugging purposes; this must be true independent of the binding time for that code.

(b) To support execution of incomplete programs. A consequence of some of the linguistic issues discussed above is that decisions (i.e., code to implement them) will be deferred as long as possible. This must not preclude compilation and testing of portions of a program which do not depend on earlier decisions.

(c) To integrate a file system into the constructive process. In particular the file maintenance of the system must have the responsibility of maintaining the structure of programs, the correspondence between different representations of the same program, keeping track of cross-references between files, distributing information from modules to compilers, etc.

SUMMARY

We have attempted to outline the need and goals for the multiprocessor computer system being constructed at CMU. The hardware and software structure were presented in overview form, together with detailed analysis of various critical parts. We believe that such a system is one which will become important in the future, simply because of the capabilities it provides and the way in which it utilizes technology.

ACKNOWLEDGMENT

A significant fraction of the faculty, students, and staff, of the Department of Computer Science at CMU are either directly or indirectly involved or have made significant contributions to this project. It is as difficult to give them all full credit as it would be incorrect to assume the authors are the source of all ideas or work reflected in this paper. Those most directly involved have been:

Professors Allen Newell and Raj Reddy who provided most of the initial motivation and served in continuing review; Bill Broadley, the manager of the engineering lab, who has designed and is constructing the specialized hardware; Chuck Pierson, who has responsibility for coordinating the project; Ellis Cohen, Roy Levin, Bill Corwin, and Fred Pollack who are programming the

operating system; Anita Jones, whose insights lead to much of the operating system philosophy; Professor Jack McCredie who has developed analytic models for the memory interference and lock problems, and Professor Mary Shaw who is developing the programming system described in the last section.

REFERENCES

1 C G BELL R CADY H McFARLAND
B DELAGI J O'LAUGHLIN R NOONAN
W WULF
A new architecture for minicomputers—The DEC PDP-11
SJCC 1970 pp 657-675

2 C G BELL P FREEMAN et al
C.ai: A computing environment for AI Research—Overview, PMS, and operating system considerations
Department of Computer Science Carnegie-Mellon
University May 1971

3 C G BELL J GRASON S MEGA
R VAN NAARDEN P WILLIAMS
The design, description and use of the DEC register transfer modules (RTM)
IEEE Transaction on Computers May 1972

4 C G BELL A N HABERMANN J McCREDIE
R RUTLEDGE W WULF
Computer networks
Computer Science Research Review Carnegie-Mellon
University 1969

5 C G BELL A NEWELL
Computer structures
McGraw-Hill Book Company 1971a

6 C G BELL A NEWELL
Possibilities for computer structures, 1971
FJCC 1971b

7 M CONWAY
A multiprocessor system design
Proceedings of the IFIP Congress Yugoslavia 1971a

8 *DEC PDP-11 documents*
Programmer Reference Manual and Unibus Interface Manual

9 E DIJKSTRA
Cooperating sequential processes
In Programming Languages F Genuys (ed) Academic Press 1968

10 E DIJKSTRA
Structured programming
Software Engineering October 1969 Rome

11 R KRUTAR
Personal Communication 1971

12 D McCRACKEN G ROBERTSON
C.ai (P.L)—a L* processor for C.ai*
Department of Computer Science Carnegie-Mellon
University Pittsburgh 1971

13 J McCREDIE
Analytic models as aids in multiprocessor design
Department of Computer Science Carnegie-Mellon
University Pittsburgh 1972

14 D L PARNAS
On the criteria to be used in decomposing systems into modules
Department of Computer Science Report Carnegie-Mellon
University Pittsburgh 1971

15 W D STRECKER
An analysis of the instruction execution rate in certain computing structures
PhD Dissertation Carnegie-Mellon University ARPA
Report 1971

16 W WULF
Programming without the goto
Proceedings of the IFIP Congress Yugoslavia 1971a

17 W WULF et al
A software laboratory: Preliminary report
Department of Computer Science Carnegie-Mellon
University Pittsburgh 1971

18 W WULF et al
Bliss reference manual
Department of Computer Science Report Carnegie-Mellon
University Pittsburgh 1971

19 W WULF D RUSSELL A N HABERMANN
Bliss: A language for systems programming
Communications of the ACM December 1971

C.ai—A computer architecture for AI research*

by C. GORDON BELL**

Carnegie-Mellon University
Pittsburgh, Pennsylvania

and

PETER FREEMAN

University of California
Irvine, California

INTRODUCTION AND MOTIVATION

A recent article analyzes the need for and possibility of an ultimate computer.[1] While the ultimate machine is still distant, much current research on system structures has the goal of significantly increasing computing power along one or more dimensions (e.g., processing speed, memory size, functional capability, reliability).[2,3,4,5,6]

The most obvious way of increasing power is through parallelism. The earliest proposal to achieve parallelism is the coupling of multiple processors to a shared primary memory—multiprocessing. Yet, the parameters of a design, especially the connections between processors, memories, and the outside world, can take on many different values.

The architecture presented here is intended to increase the computing power available for a particular application—artificial intelligence research. It was formulated under the constraint that if built, it would have to be operational within two years. Its lifetime was assumed to be on the order of five years, with a slow rejuvenation replacement process occurring during use. Although it was formulated to be used via a network such as that of the Advanced Research Projects Agency (ARPA), the result has implications for the design of any currently feasible, very large computer.

We present two major parts of the design—the structure of the hardware (and some of its details*) and the operating system—as originally stated** and then briefly discuss its implications for research on multiprocessing.

Requirements for ai computing

A number of special function computers ranging from business to scientific have been described.[8] The characteristics of machines used for ai research appear to span this spectrum, exhibiting the maximum of each characteristic attribute. The more important of these attributes will be examined from the standpoint of ai computing.

Memory size

The primary (program) memory is larger in an ai environment than most other computers because of the local program and data base. Usually data is highly interrelated and linked for such programs as natural language processing. We have assumed an average program size of 250,000 74-bit words and a working-set size of 100,000 74-bit words. The ratio of secondary memory (e.g., drum) to primary memory (e.g., core or integrated circuit) we have proposed is quite low, based

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

** Present address: Digital Equipment Corporation, Maynard, Massachusetts.

* The PMS notation[7,8] used throughout this report is based on seven primitive component types: P-processor, M-memory, S-switch, L-link, K-controller, T-transducer, D-data operation. A computer composed of primitive components is represented by C; hence, C.ai for "ai computer."

** The design was first presented in a technical report[9] in May, 1971.

more on swapping operation than demand paging. This appears justified since the list structure of many of the programs may not permit small working-sets, although a better model of a typical program is clearly needed. A coupling to a larger tertiary memory is provided for files.

Processor power

The central processor power requirement for ai computing seems less than for the largest scientific processor because the typically large memory is accessed more or less randomly in a relatively inefficient fashion. Thus, the more powerful facilities of a scientific processor may go unused because the data-types (e.g., floating-point) are unused. In some cases ai processes are compute-bound, and need significant processor power in critical areas.

PMS structure

The ai computer usually requires better communications facilities to external equipment and humans than either a scientific or a business computer. Typically, these characteristics are like the process control computer, except that higher data rates are involved for speech, video and mechanical transducers which operate at human interaction rates. These considerations are particularly critical for ai research groups engaged in robot, hand-eye, and speech research.

The instruction set (conventional and specialized)

The design permits the use of existing conventional processors as well as specialized processors. Conventional processors can be either 32 or 36 bits. In particular, since the PDP-10 is widely used for ai research, PDP-10 processors can be used (e.g., DEC's KA10 and KI10, and the Stanford Artificial Intelligence Laboratory version).

There is also a need for a facility which can be used for experimentation with new instruction-sets for ai processing. Specialized language processors can be fabricated, tested, and used within the environment. Typically, the instruction-set is the characteristic that usually comes to mind when ai computing is discussed. Operations for a stack, a garbage collector, hash-coding on linked lists, etc., are obvious candidates. On the other hand, a simple processor with floating point arithmetic is often adequate because decisions can be bound in software and later changed.

FUNCTIONAL OVERVIEW OF C.ai

Consideration of the problem of designing and building an optimal computer for ai research quickly leads one to the realization that there may not be a feasible solution. The numerous constraints, wide variations in computing style, and the impossibility of defining the ai problem narrowly seem to make this a certainty. Thus, the major premise of this design is that if one wishes to provide ai researchers with better computing tools, one must, in fact, provide an *environment* in which many, varied tools may be developed and used. This design should be viewed as a specification of such an environment.

The aim is to provide a collection of virtual machines, organized along multiprocessor lines. It is thus important to understand what the system provides and what the users see.

The user sees

- a collection of functionally specialized interconnected computers, each with large memories;
- the potential for processes on separate processors to communicate and share resources;
- a large secondary memory for temporary use and a very large tertiary memory (i.e., files) for permanent storage of information.

An operating system on a processor sees

- a mechanism for allocating and overseeing sharing of the three level memories (primary, secondary and tertiary);
- a mechanism for the transfer of files between memory levels;
- a mechanism to handle the transfer of information to the outside world.

The overall operating system sees

- processors competing for memory;
- requests to share resources between processors;
- logical communication channels between processors and the outside world;
- files to be created and moved;
- memories to housekeep;
- accounting information to be logged and displayed.

THE HARDWARE STRUCTURE OF C.ai

Figure 1 shows a simplified diagram of C.ai. It is a multiprocessor system with a centralized cross-point

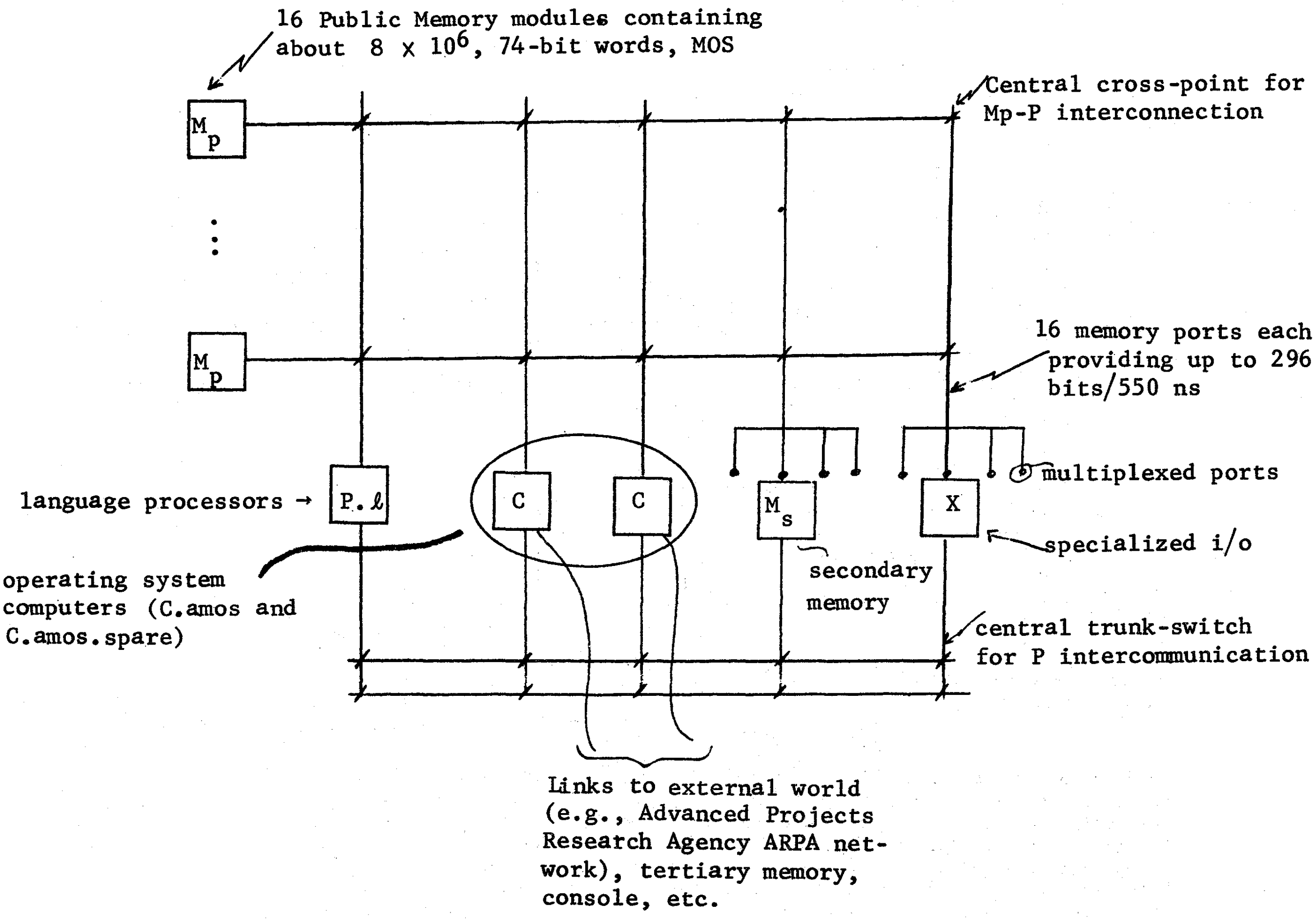

16 Public Memory modules containing about 8 x 10^6, 74-bit words, MOS
Central cross-point for Mp-P interconnection
Mp
Mp
16 memory ports each providing up to 296 bits/550 ns
multiplexed ports
language processors →
P.ℓ
C
C
Ms
X
specialized i/o
operating system computers (C.amos and C.amos.spare)
secondary memory
central trunk-switch for P intercommunication
Links to external world (e.g., Advanced Projects Research Agency ARPA network), tertiary memory, console, etc.

Figure 1—PMS diagram (simplified) of C.ai

switch which allows up to 16 processor or secondary memory (e.g., drum) ports to be connected to the primary memory. Some ports can be multiplexed between several components. The characteristics of the primary memory (the dominant component of the system) are: MOS, 550 ns cycle time; 2^{21} 296-bit words accessible as 74, 148, 222 or 296 bits in a single access.

The 16 memory modules are interconnected to the 16 processor ports via a central 74-bit cross-point switch. The main reason for a central cross-point is to limit the cables from $p \times m$ to $p+m$ and shorter distances. Also, by using a non-bus arrangement, processor and memory modules can be removed while the system is operating.

Another type of switch is used to provide intercommunication among the processors. This is shown in the lower part of the figure and is described later in more detail.

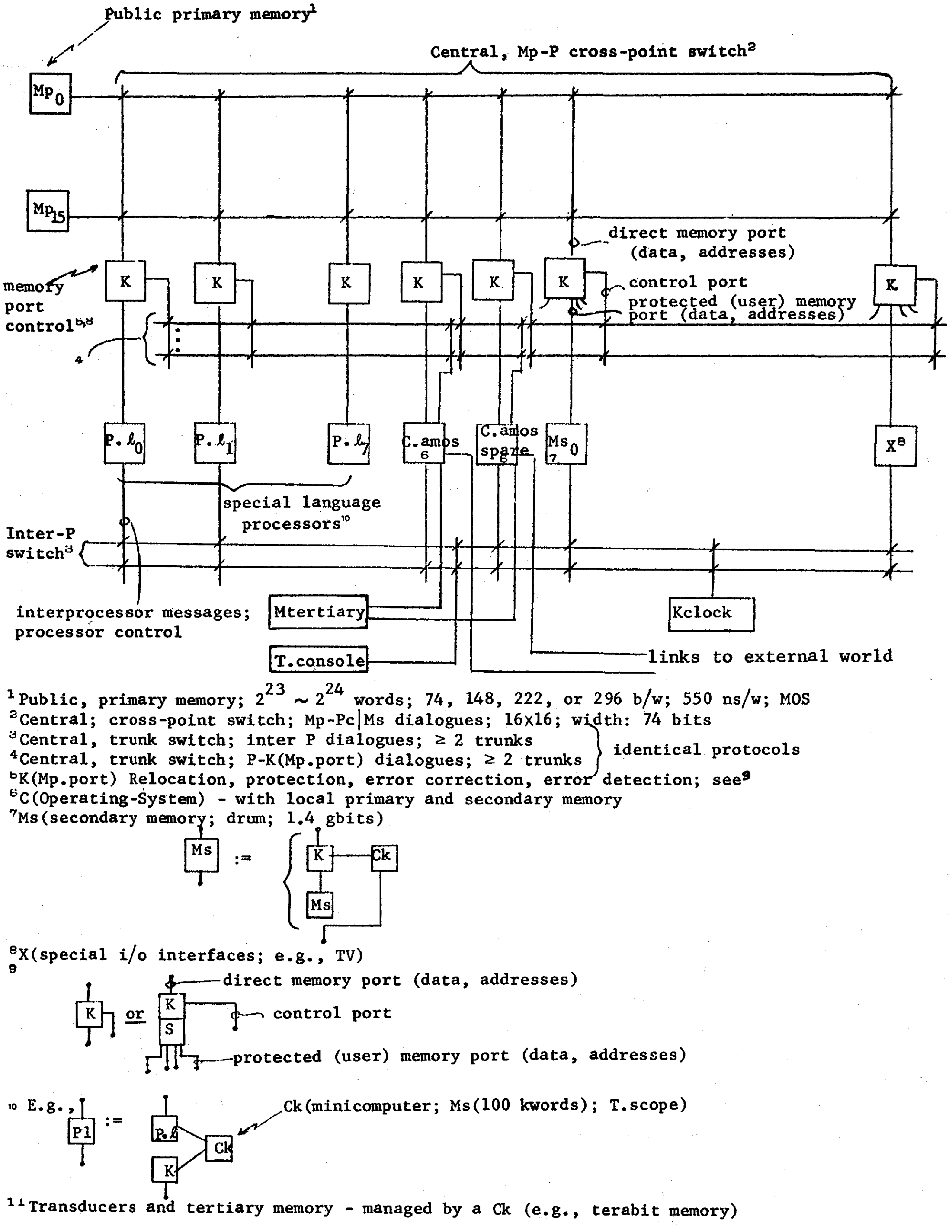
A central computer (C.amos) executes the operating system. It has some private memory, interconnection to i/o devices, terminals, and tertiary memory. Two C.amos computers are shown, but only one will be in use at any time (see below). The other processors are general-purpose or specialized language processors. Each of these might have a minicomputer which acts as the control for starting and stopping, maintenance, data gathering, context switching, etc.

Figure 2 provides a more detailed description of the structure shown in Figure 1. It will be used to aid the description.

Primary memory

The characteristics of various memories are given in Table I. We have used specific quotations for cost, performance, etc.

The primary memory, Mp, gives an overall memory size of about 620×10^6 bits and (because of the 16 ports) a bandwidth of 8,600 million bits/sec (16 ports $\times$ 296 bits/port; .550 μsec cycle). The access time, as measured at the processor, will be about 350 ns. The actual


Public primary memory[1]
Central, Mp-P cross-point switch[2]
Mp0
Mp15
direct memory port (data, addresses)
memory port control[5,8]
4
K
control port
protected (user) memory port (data, addresses)
P.l0
P.l1
P.l7
C.amos 6
C.amos spare 6
Ms 7 0
X[8]
special language processors[10]
Inter-P switch[3]
interprocessor messages; processor control
Mtertiary
T.console
Kclock
links to external world

[1] Public, primary memory; $2^{23} \sim 2^{24}$ words; 74, 148, 222, or 296 b/w; 550 ns/w; MOS

[2] Central; cross-point switch; Mp-Pc|Ms dialogues; 16×16; width: 74 bits

[3] Central, trunk switch; inter P dialogues; ≥ 2 trunks } identical protocols

[4] Central, trunk switch; P-K(Mp.port) dialogues; ≥ 2 trunks

[5] K(Mp.port) Relocation, protection, error correction, error detection; see[9]

[6] C(Operating-System) - with local primary and secondary memory

[7] Ms(secondary memory; drum; 1.4 gbits)

[8] X(special i/o interfaces; e.g., TV)

[9]

K
or
K
S
direct memory port (data, addresses)
control port
protected (user) memory port (data, addresses)

[10] E.g.,

[11] Transducers and tertiary memory - managed by a Ck (e.g., terabit memory)

Figure 2—PMS diagram of C.ai

memory word will be 296 bits with a processor specifying which word (74 bits) or configuration of words it wants. The actual information bits are expected to number 64 with 10 bits to be used for error correction or detection at the processor. Alternatively, the 74 bit word will allow present 36-bit processors, such as the PDP-10, 1108, and Honeywell 645 to utilize the memory. The memory will consist of 16 modules and will connect to the processors through a 74-bit wide cross-point switch. There are 16 processor ports each 74 bits

TABLE I—C.ai Memory Characteristics

Device	function***	t.access (t.cycle)	i.rate mb/s	i-width (bits)	total i.rate (mb/s)	power (watts)	size module 10^6b	cost (k$)	cost/bit ¢/b	controller costs (k$)
photomemory	t						$\geq 1\times 10^6$			
moving-head disk	t	0~55 + 20 ms	3.3**	1×3	3.3×3	15 kw/ unit	200×20=4000	300	0.0075**	300
drum	s	8 ms	3.3	8~16/drum	50/drum	1 kw/drum	70×20=1400	900	0.048	5×40
shift register	s	131 µs 780 µs	1	296	296	200 kw	1400	1800	1.25 0.7	5×40
core	p	.7 µs (1.8 µs)	.55	296×16	163 2605	200 kw	620	10,240	1.65	500
MOS	p	500 nsec (1.2 µsec)	.83	296×16	246 3947	200 kw	620	5760	.93	500
MOS	p	250 ns (400 ns)	2.5	296×16	740 8600	200 kw (.2 mw/ bit)	620	9500*	1.52	500
Bipolar	i	40 ns (80 ns)	10	296	2960		100~2000 words	36	6	
Bipolar ROM	i	40 ns (80 ns)	10	50~296	2960		100~2000 words	12	1~2	

*Estimate

**These assume current densities. We can safely assume double density, hence lower cost, more storage, and higher transfer rates.

***t/tertiary; s/secondary; p/primary; and i/internal processor (cache, program control, accumulators, etc.)

wide. A transfer of more than one 74-bit word in an access will be done sequentially at a rate of 75 ns per additional 74 bit word for up to four words.

Processor port control—K(Mp.port)

The local port control is shown in Figure 3. Each processor port provides access to 2^{24} words (a 24 bit address). The upper 7 bits of the address specifies one of the 128 relocation (mapping) registers the address will use. The relocation register will supply the high order bits of the physical address and the processor address will supply the low order 17 bits. This is a concatenation, instead of an addition, and thus should be quite fast ($\leq$50 ns). The relocation (mapping) registers and other controls associated with the port are accessible only to the overall operating system. Various protection-type bits might also be included in the memory port control box to assist the processor. The relocation unit serves these functions: maintenance, dynamic memory assignment (reconfiguration), protection and sharing among processors, and data parity checking.

The relocation registers will be transparent to each processor and will serve the function of manual, address switches on the memory. Thus, no manual switching need be provided on the individual memory modules; the same effect is achieved by informing the memory control processor to vacate the desired module and consider it unusable. All relocation and protection, as commonly found in timesharing system processors, will be included as part of a processor. The only reason a processor might want to consider the port relocation registers is to effect 65k word block transfers, i.e., to ask the memory control processor to change addresses, e.g., 128k-192k to 64k-128k.

The statistics control shown is passive. Although not detailed here, it will be connected to provide appropriate information on accesses, errors, and transfer rates to a measurement unit. Another part of the port interface is the capability of being exercised at low data rates via the controlling computer. Thus, data can be transferred via each port from the control computer. Within each port control there is error correction and detection hardware. Here, since we assume that some faulty processors will be attached to the ports, the port control

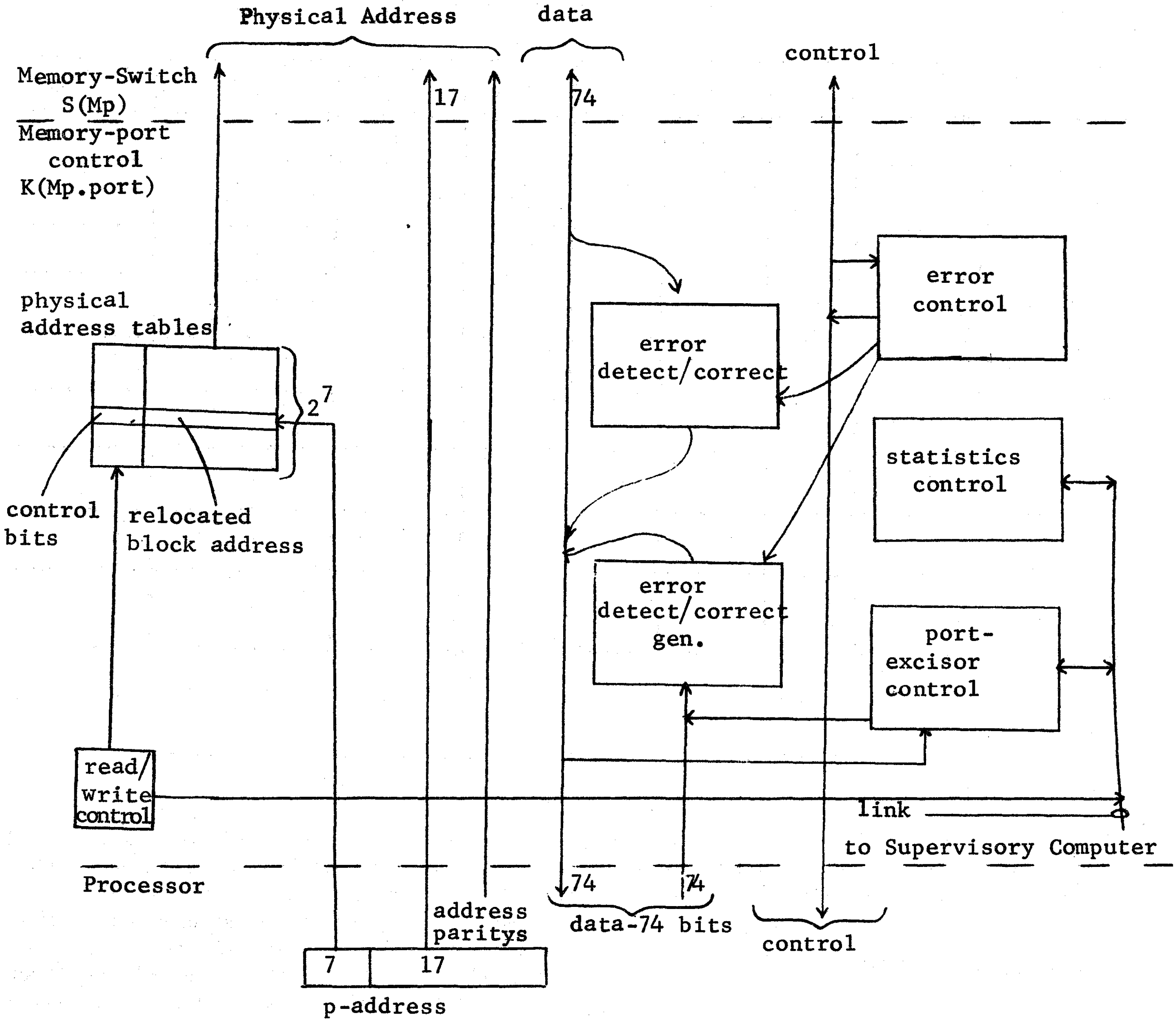


Figure 3—K(Mp.port) memory-port mapping (relocation), error-detection, error-correction and control

must supply correct data to the memory in order that other devices (e.g., drums, disk memory) can detect faults.

Primary memory switch

Since the switch is critically central, a dual cross-point may be preferable. This is essentially a compound switch consisting of two cross-point switches and a (m+p)×S(1-input, 1-output) switch as shown:

Mp *m-inputs*

S(cross-point; m×p)
S(m; 2) S(p; 1) *p-inputs* P
S(cross-point; m×p)

Current logic technology is ideally suited to the packaging of a centralized switch. Although it is centralized, the physical packaging can be carried out to provide independence among the processor and memory ports. Logically, the memory controls and the processor controls are quite independent. By partitioning the switch into four 8×8 switches, even more independence can be gained.

The memory switch can utilize current MSI (medium scale integration) logic. The switch will utilize 16 bit multiplexors with a typical data propagation time of 10 nsec (assuming the data select lines have been settled for ≥ 30 nsec). Faster circuits, such as Schotky TTL and LSI switching modules, will probably be available in time for any actual construction.

Secondary memory

The secondary memory investigated includes: (1) mechanical devices (drums, fixed head discs, etc.), (2, shift register or other block-oriented solid-state memory, (3) and random access memory (RAM). Current characteristics for these devices are shown in Table I. Mechanical devices will probably hold a price advantage of an order of magnitude for several years. It does not appear that shift register memory will become sufficiently cost effective over random access memory for our purposes. On the other hand, block-oriented random access memories may be available shortly.

A secondary memory system might consist of 20 drums, for example, with the characteristics given in Table I. Such a system would give 1,400 megabits of storage, an average access time of 8 msec, and a transfer rate of 50 megabits/second.

Initially, the secondary memory controllers will simply permit multiplexing several drums into one port. An additional feature that could easily be included in the secondary memory channel would be a memory-to-memory connection that could take advantage of the 4-word sequential feature of the primary memory. Since one wishes to maximize the bandwidth between secondary and primary memory, as the system grows to use many drums on several ports it will probably be necessary to insert a computer to control the secondary memory, C(Ms). The secondary-tertiary memory system might then look as shown in Figure 4.

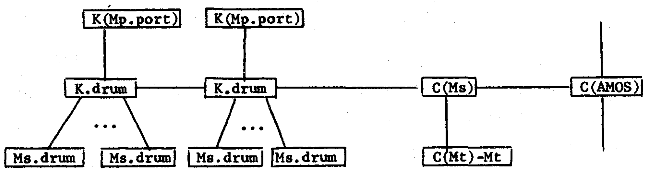


Figure 4—Eventual secondary-tertiary memory structure

Tertiary memory

Clearly, a computer of this capacity requires some on-site mass storage. This will permit programs to reside on tertiary memory until they are brought into either primary or secondary memory for more rapid access. The tertiary device will be controlled by its own processor. Aside from its size (on the order of 10^{12} bits), it has not been specified any further.

Console

Scopes will be used to display the overall allocation of resources to tasks, and the status of each processor and the overall system. Several scopes may also be employed for human intervention required in the management of the system.

Interprocessor communication

Interprocessor communication will be carried out over a data bus similar to DEC's PDP-11 Unibus, with the exception that it would be a dual or multiple trunk bus to increase bandwidth, decrease response time and increase reliability. A processor making an interprocessor transfer would place a request on the bus and the actual transfer would take place on the trunk that first responded. Each message will be tagged with the identity of the transmitting processor. A processor will be able to communicate with itself on the bus.

If the proposed interprocessor traffic appears to warrant it, more than two trunks can be added. However, processors may communicate at high data rates through shared primary memory.

A SOFTWARE STRUCTURE FOR C.ai

This section provides both an overview and first level design of AMOS, a minimal operating system for C.ai. The system is not specified completely, however.

In a system with multiple active units (processors, in the case of C.ai) and shareable resources there is a spectrum of possible systems ranging between the extremes of having all control of resources vested in a single active element to having no distinguished component with respect to resource allocation. AMOS is a classical design in which ultimate control of all shared resources is by a single component although all nonshared resources (e.g., processors*) control themselves.

* Processors may, of course, be shared among processes on a local basis. Our concern here is with the global management of the system.

Likewise, there is a spectrum ranging from the highly uniform in which the user is unaware of the existence of multiple components working on his task to the highly diverse in which the user of one component is unaware of the existence of the others. AMOS tends toward the latter extreme. The hardware architecture does not prevent the former, however.

Design goals and guidelines

While not thoroughly defining the space of systems we are interested in, the following provide a partial specification:

1. Time and effort needed to construct AMOS must be small.
2. The functions provided by AMOS must be minimal, consistent with managing the hardware resources of C.ai.
3. The "users" of AMOS are the operating systems for each processor. Thus the total operating system is a two-layer object: an overall operating system (AMOS) plus distinct operating systems on each processor. In most cases a human user and/or his program sees only one of the individual systems, not AMOS.
4. Specification and construction of the operating system for a processor is the responsibility of its designer.
5. AMOS should usurp as few design prerogatives as possible. That is, it should influence only minimally the design of operating systems and programs on individual processors. Further, it should not greatly influence the design of C.ai as a whole so that in the future it will be possible to replace AMOS with a completely different operating system.*
6. It should be possible to build very simple operating systems on the processors if desired. They should not have to handle transfers to i/o devices and their communications with AMOS should be simple.

Functions to be provided

It is easiest to specify what AMOS is to do by listing the major functions it is to provide. Elaborations of these functions will be provided below in describing their implementation. It is assumed that a few other minor functions will be needed and can be added without greatly perturbing the design of C.ai or AMOS.

1. Allocate primary memory. Individual processors must be given access to varying amounts of main memory. Addressing ranges and access protection must be set.
2. Allocate and control other on-site memory. Secondary and tertiary memories must be allocated, but control must remain with AMOS in order to enforce security of parts allocated to different processors (and processes).
3. Handle communication between processors and the external world. In order to keep the operating systems on the processors simple, communication must be handled by AMOS.
4. Provide system status and accounting information. An on-site console must be maintained in addition to logging accounting information.
5. Startup of C.ai and individual processors. Occasional cold starts of the entire system will be necessary. Individual processors may come up and go down as well.
6. Movement of files between memory levels. Processors should not have to deal with physical i/o. Further, large stores must be a shared resource.

Structure providing the required functions of AMOS

Different structures can be chosen to provide the functions of AMOS. Those selected below seem to be sufficient for the task and consistent with the design objectives. A more detailed overall design and/or simulation may, of course, indicate the choice of alternative structures.

Primary memory allocation

The opaqueness of how allocations of primary memory are being used and their size (64K words) implies using an extremely simple algorithm. A processor will send a request to AMOS over the bus to allocate or deallocate a page of Mp; the request will include where in the processor's address space the page is to go. AMOS will check whether or not the processor is entitled to another page (a policy decision) if a new one is being requested. It will then adjust the mapping of the processor appropriately and signal it that the allocation has been made.

In order that several processors be able to handle

* C.ai is clearly a unique opportunity for implementingr adically new virtual machines that exploit its parallel and functionally specialized parts. The understanding of such a machine, how to break up a load computationally, the characteristics of the programs run on it, etc., is so meager that initially the only sensible way to use it is as a collection of independent systems that happen to share some physical resources. AMOS and its hardware should not unduly impede research on more advanced modes of usage, however.

large jobs at the same time, it will be necessary for the operating system on each processor to release primary memory on a second-by-second basis whenever it is free. This might be handled on a gentlemen's agreement basis with perhaps some monitoring in AMOS to insure that no processors use too much core. The specific algorithm to use is a policy decision.

Secondary and tertiary memory

As far as a processor is concerned the basic unit of storage will be an arbitrary length file. (For efficiency, information actually may be stored on secondary and tertiary memory in standard block sizes.) A processor can request AMOS (via the communications bus) to create a file; AMOS will check if the processor can have more space and if so, create a name for the file and pass it back. (To facilitate storing, the names should be from a single continuous space.) A processor can request information to be transferred among memories.

The request can be made with a priority, thus allowing swapping or paging information to be handled just like any other file only with higher priority for performing the transfer. Likewise, files can be transferred from secondary or tertiary memory to primary memory. Alternatively, external information can be transmitted directly to or from a file (see below). Files can be erased by request.

Note that the processors specify where they want their files to reside. This seems essential since only they will know the use. Pricing structure, time limits, and allocation limits can be used to insure proper migration.

It is assumed that lower level memories provide hardware detection of record and file ends so that transfers of partial files may be made. On the other hand, record transfer may impose too much additional complexity on AMOS.

Communication to the outside world

AMOS will know nothing about specific users. It will have only logical channels that it can connect between a processor and some external entity transmitting messages to C.ai. Since C.ai is intended to be a resource for use among a large number of users via a network (in this case the ARPA network) this will provide the mechanism for establishing contact between users and their processes.

AMOS may receive messages from entities for which it has no logical channel set up, requesting access to a given processor. The processor may have told AMOS that it will take all callers, only certain ones, or that it wants to be informed of all requests for connection so that it can make a dynamic decision. If the requestor cannot be attached, he will be so informed.

If a user can sign on, he is given a unique identification by AMOS and a logical channel is established to the desired processor. Until the connection is broken by the processor any incoming information headed by that identification will be sent to the proper processor (deposited in a section of his Mp or on one of his Ms files) with a signal going from AMOS to the processor whenever a transmission is completed.

A mode will be available for the transfer of large blocks of data directly to a secondary or tertiary memory file without interrupting the processor until the transfer is finished (even if it takes many transmissions). Similarly, a processor can request a file of information of any size to be sent out over a given logical channel.

System status and accounting

AMOS will record all system resource usage (e.g., memories, i/o gear, external links) by each processor. The information will be displayed in summary form on a console and made available to the processors if appropriate. A processor can access the data of another under the usual sharing rules (see below). It is up to individual processors to record their own usage and to subdivide their use of system resources among their various users.

Each processor will supply a certain amount of status information to AMOS upon request in order to produce system-wide status displays. The content of this information depends on more detailed specifications of how individual processors will be used.

Any error checking or internal monitoring information available to AMOS will be displayed appropriately. AMOS will also be responsible for utilizing such information to warn of faulty components or potential system bottlenecks.

Initialization

C.amos will have an autoload button that will load its local memory from a start-up disk with a program to initialize Mp bounds registers and load its main Mp from Ms. Its bootstrap will also be able to retrieve from its local Ms various debug, checkout, and recovery routines.

C.amos will be able to start up any of the other processors by a signal over the bus. Once started, however, AMOS has no control over the processor. This means that AMOS will have available the operating system (or a bootstrap) for each processor. In some cases this may include loading a microcode store.

File movement

All file operations are logical (not physical) as described above. AMOS will have one or more minicomputers that will initiate transfers between memory hierarchies and perform housekeeping chores.

Resource sharing

The mechanism for sharing is basically the same for all resources. Processor A (the owner of a resource) tells AMOS over the bus that processor B may have a given type of access to the resource. If later, B requests that access, it will be granted (unless A has rescinded the access rights).

In the case of Mp this is implemented by setting bounds registers. For files AMOS must keep lists of processors (*not* processes) that can access given files. It is then up to the processors to control the access of their individual processes.

As an example of primary memory sharing, A may tell AMOS that B can have read access to one of its pages, say P. A will communicate directly to B that it has permitted access to P. Later, if B requests access to P, AMOS will set one of B's bounds registers to permit read sharing. Permission for sharing (or relinquishing by B) can occur at any time. A and B must insure that permission is not withdrawn precipitously.

Performance monitoring

A computer such as C.ai must have adequate performance monitoring capabilities integrated into its basic design. Many initial decisions will require modification as the system matures and usage patterns evolve. Proper design of memory systems, optimal allocation of primary memory, and correct bandwidth to the outside world are examples of decisions requiring extensive measurement of usage.

Measurements should occur on a number of levels. Common facilities such as the interprocessor bus and the primary memory could be monitored by passive hardware devices connected to a separate computer, C.pm. This independence would insulate the C.pm from changes in AMOS and in the processors. C.amos and C.pm could communicate directly for dynamic control, but other information would be stored for later analysis. Many items can be measured passively by: (a) busy-idle bits; (b) registers to read or sample; (c) counters. AMOS must be able to interrogate C.pm to obtain current data for scheduling and resource allocation and for system status requests from processors. Thus, the C.pm should have the following features:

(a) Pc-Mp-Ms (processor-primary memory-secondary memory) type of structure;
(b) ability to reduce its own data and keep current system information available for AMOS;
(c) ability to write to its own slow Ms for later display and analysis.

Some examples of information of interest are: (a) K(Mp.port) errors could be counted, and transmitted to C.amos, (b) the number of memory references could be counted and waiting times tabulated, (c) a central clock may be provided which all processors may access. A central timing facility might also be included at the clock. In order to keep the traffic low, a facility such as the clock might broadcast the time so that each processor could maintain its own timers (which would undoubtedly be in software).

AMOS should have a number of software monitors built into its modules. Such hooks are best when implemented in parallel with the operating system. Selected information would be either written by AMOS or read from registers by C.pm. If AMOS is to be a resource allocator, some processor information may be required. Information of this type would place certain constraints on processor implementors, but the sharing of common resources requires some standardization.

Each processor should also include its own hardware and software measurement devices with which AMOS can communicate. A mixture of software and independent hardware monitors integrated into the design of C.ai will allow for future study of this new structure, and encourage growth based on a knowledge of actual performance and utilization.

PERFORMANCE CHARACTERISTICS AND EVALUATION

Table II shows a comparison of C.ai performance with some current large-scale computers. Various attributes of these machines are given in order to give the reader an idea of the balance of the computer in terms of memory size, processing capacity and cost. The measures used by Roberts[10] were included to compare the performance with these machines. In some cases the chart is misleading since C.ai has 20 times the memory of the next largest machine (STAR). However, for ai research, memory size is probably the single most im-

TABLE II—Comparison of C.ai With Other Computers

	Mp.width (b/w)	Mp.size (mwords)	Mp.size (mbits)	Mp.i-rate (mwords/sec)	Mp.i-rate (mbits/sec)	Ms.i-rate (mbits)	Pc.i-rate (mop/s)	Cost(m$);Bit×mops/ sec;^{10} Bit×mops/sec/$
PDP-10	36+1	0.26	9.7	4	144	9	0.4	1; 14.4; 14.4
Stanford AI-10	144+4 (36b/instr)						4.0	1; 144; 144
Model 91	64+8	0.52	37	21	1370	10×(1~2)	6.0	7.7; 384; 49.9
CDC 6600	60	.26	15.4	32	1920	12×10 (i/o) 600 (ECS)	3.0	5.5; 180; 32.7
C.ai	74~296	8.3	620	29~120	2150~ 8600	50×5	(4~12)×10 (4~12)×20	13*;1440~4320; 110~330 16;2880~8640 180~540
C.ai/4	74~296	2.3	155	8~30	504~ 2150	50×2	(4~12)×3 (4~12)×6	
CDC STAR	512 (32b/instr)	.131	74	25	12,800	(50~100)× (1~5)	100	10; 3200; 320
ILLIAC IV[5]	64+	.131	8.2	64×4.1= 261	16,800	1000	256	10; 16384; 1638.4

*Assumes $8m for memory, $2m for peripherals and $300K per processor (10 Pc's in first case, 20 in second); Stanford AI-10 assumed. Adjusting the memory size to that of STAR, yields $7m (total); 1440~4320; 205~620 and $10m; 2880~8640; 288~864.

portant characteristic. This has been adjusted in the footnote to the table. The computation is based on 36 bit operations. Using a larger word would increase the performance indicator, though probably not any real performance for this task.

The PMS diagram of Figure 2 has sufficient detail for deriving basic performance characteristics of the computer. Each processor is assumed to cost approximately $100,000. A 16×16 switch should run approximately $200,000.

The critical parameter for determining the performance is the number of memory ports and the processor operation-rate, so that the interference among the processors can be determined. Each processor is able to obtain up to 296 bits in 550 ns (or 540 megabits/sec). By comparison, a PDP-10 demands roughly two words each 3.5 μs at 300,000 op/sec. Thus, a word can be used each 1.25 microseconds or its port needs only 28 megabits/sec. The above system supplies roughly 20 times this amount; eight words/access and a cycle time of 550 ns contribute to this gain. Since the eight words are accessed at one time, a cache next to a processor will be necessary to make full use of the capability.

By using a cache memory the needed bandwidth into primary memory is significantly reduced. One would expect about 95 percent of the data in the cache. Executing 4 million instructions/sec, at most, one word would be required per 125 ns (i.e., 8 million words/sec). Now since 95 percent of the data is in the cache*, the requirements for primary memory access are only one access each 20 memory accesses. Thus the effective memory cycle time is only one word each 2.5 microseconds. The interference among ports is therefore quite small. At the very least, i/o devices, such as the drums, could share a port. In Table II we have assumed two processors per port (for a total of 20 processors).

CONCLUSIONS

An overall argument has been given as to the feasibility and desirability of building a computer to be used in ai research. The design is a very conservative, simple approach built on current computers and technology. Only conventional performance processors were assumed (i.e., each has about the same performance as a 360 Model 85).

Given the overall results, this design provides a basis for specification of the next level of detail. We believe that an approach that departs from a conventional structure (e.g., by placing specific interpretation on ad-

* Cache simulation for LISP interpreter.

dressing) will both decrease the performance and also make the memory too specialized, thereby eliminating unspecified future use that might be made of such a large facility.

The real emphasis of the structure is simplicity, yet it provides much potential power (bandwidth). Also, the design is not presumptuous about how particular future processors will use the facility. In particular, additional power can be gained by developing special purpose processors to be used on C.ai.

Although there are no plans to implement C.ai, a project at Carnegie-Mellon University, the C.mmp multiminiprocessor computer[11] has a similar architecture. Indeed, C.ai has already influenced the structure and instigation of that project. C.mmp is being fabricated and should provide concrete operational evaluation of the design proposed here, particularly the central switch, processor intercommunication, and operating system. The C.mmp machine is being built to provide computing power for speech processing and is thus more than an architectural research experiment.

Multiprocessing is often taken in a rather limited sense, but it can encompass a range of computing modes: parallel processing, pipelining, networking, functional specialization, and independent but cooperating processors. The simplicity of C.ai and the fact that the ai computing environment is so general makes this design well-suited to support research into the various forms of multi-processing.

Just as experimental observations of the physical world and theorems are presented for their own merit, we believe that system designs should be made known and studied as a source of ideas for other designs. Hopefully the architecture presented will serve this purpose.

ACKNOWLEDGMENTS

The design reported here was developed in a project seminar on list-processing machines run by the authors at Carnegie-Mellon University during the Spring of 1971;[9,12,13] the participants worked on the design at all levels and their contributions are acknowledged. Valuable assistance and feedback was provided by Professors A. Newell, R. Reddy and W. Wulf. A Kendziora and Professor J. McCredie provided the section on performance measurement. The referees' suggestions greatly reduced the size and improved the organization and readability.

REFERENCES

1 W H WARE
The ultimate computer
IEEE Spectrum March 1972 p 84

2 C G BELL R CHEN S REGE
Effect of technology on near term computer structures
IEEE Computer March/April 1972 p 29

3 D J FARBER K LARSON
The structure of a distributed computing system software
Proceedings of XXII Polytechnic Institute of Brooklyn Symposium April 1972

4 M J FLYNN A PODVIN
Shared resource multiprocessing
IEEE Computer March 1972 p 20

5 J H BARNES R M BROWN M KATO
D J KUCK D L SLOTNICK R A STOKES
The ILLIAC IV computer
IEEE Transactions on Computers C-17 Vol 8 p 746 August 1968

6 S A HOLLAND C J PURCELL
The CDC STAR-100: a large scale network oriented computer system
Proceedings of the IEEE Computer Conference September 1971 p 55

7 C G BELL A NEWELL
The PMS and ISP descriptive systems for computer structures
SJCC 1970 p 351

8 C G BELL A NEWELL
Computer structures
McGraw-Hill 1971

9 C G BELL P FREEMAN et al
C.ai: a computing environment for ai research
Computer Science Department Carnegie-Mellon University Pittsburgh Pennsylvania May 1971

10 L ROBERTS
Data processing technology forecast
Advanced Research Projects Agency April 1969

11 W A WULF C G BELL
C.mmp: a multiminiprocessor
This volume

12 M BARBACCI H GOLDBERG M KNUDSEN
C.ai(P.LISP)—a LISP processor for C.ai
Computer Science Department Carnegie-Mellon University Pittsburgh Pennsylvania May 1971

13 D McCRACKEN G ROBERTSON
C.ai(P.L)—an L* processor for C.ai*
Computer Science Department Carnegie-Mellon University Pittsburgh Pennsylvania May 1971

Syntactic formatting of science information

by NAOMI SAGER

New York University
New York, New York

INTRODUCTION

It has been increasingly recognized that science information systems have need of natural language processing. F. W. Lancaster, author of the National Library of Medicine Study of the performance of the MEDLARS system,[1] spoke of this at the 1971 annual conference of the ACM, in the panel "Can Present Methods for Library and Information Retrieval Service Survive?"[2] He noted that "there is a definite trend away from large carefully controlled vocabularies and toward natural language processing, or at least machine-aided indexing," and quoted Klingbiel's remarks to the effect that "highly structured controlled vocabularies are obsolete for indexing and retrieval" and that "the natural language of scientific prose is fully adequate for these purposes."

In the direction of more flexible, user-oriented systems, the question has also been raised as to whether computer methods can be developed for accessing the information in scientific articles directly, without the mediation of a librarian or systems expert between the user and the stored information. Professor J. Belzer, chairman of the above panel, raised this question: "Our so-called information retrieval systems are in fact not information retrieval systems. They are bibliography producing systems, and we store documents and not information. . . ." "Were the system able to supply him (the user) with the information he wanted, it would not be necessary for him to read the entire document." In light of these remarks, we ask: Is it indeed possible for a mechanical system to identify the portions of a text which contain specific information? Can the information in sentences of the natural language text be organized on the basis of computer processing of the text so that each sentence becomes a case of a regular pattern which is both linguistic and informational, i.e., a format?

That the answer to this question is "yes," is suggested by the results of a recent research into the specialized use of language in scientific subfields. The discourse in a science subfield has a more restricted grammar and far less ambiguity than has the language as a whole. We have found that the research papers in a given science subfield display such regularities of occurrence over and above those of the language as a whole that it is possible to write a grammar of the language used in the subfield, and that this specialized grammar closely reflects the informational structure of discourse in the subfield. We use the term sublanguage for that part of the whole language which can be described by such a specialized grammar.

The sublanguage grammar provides a method for developing the particular word classes (the special-word sets) and the relations among these classes which are of special significance in a given science subfield, i.e., which are the linguistic carriers of the specific knowledge in the subfield. Yet these categories and relations are not determined *a priori* for the subfield. Rather, they are the interpretation of the formal grammatical categories and relations of the sublanguage grammar. Thus, in the pharmacological sublanguage which was investigated, the two noun subclasses I (containing, e.g., *ion*, *K*$^+$) and G (containing, e.g., *drug*, *digitalis*, *glycosides*), which in the subfield have the significance "ions" and "pharmacological agents," respectively, and play crucially different roles in the physiological mechanisms being described, are obtained as separate classes because they occur with different classes of verbs: e.g., I as the object of such verbs as *transport*, G as the subject of such verbs as *inhibit*. It then turns out that the sublanguage word classes, which are established on the grounds of what other grammatical classes they occur with (as subject, object, etc.), are the linguistic counterparts of the real-world objects, events, and relations which are studied and described in the given subfield.

A sublanguage grammar leads to a grammatical format for sentences in the sublanguage in which the words in each "slot" of the format are found to correspond to a particular kind of information in the subfield. For the pharmacological subfield whose grammar is summarized below, there are grammatical slots corresponding to: biochemical or physiological events, quantitative relations, drug actions, connections between science facts, and experimental and epistemic relations of the scientist to the objects and facts of the science. As with the sublanguage grammar itself, the words of a sentence are not assigned to the slots of the sentence format on the basis of their semantic properties, but on the basis of their subclass standing vis-a-vis other grammatical word classes in the sentence. A description of the formats for the pharmacology sublanguage and examples of formatted sentences are given following the summary of the sublanguage grammar, below.

SUBLANGUAGE GRAMMAR

The following is a sketch of the sublanguage grammar for the pharmacological subfield dealing specifically with the cellular level actions of the cardiac glycosides (digitalis).*

Location of the science vocabulary in the sentence structure

For purposes of this work, the structure of a sentence can be represented by a string decomposition obtained mechanically by a computer program,[3,4,5] or by a transformational decomposition,[6] or a transformational lattice.[7] In the latter two types of analysis, each sentence of the sublanguage is decomposed into one or more elementary sentences S_e with a succession of (partially ordered) operators which operate on the S_e or on the S_e with operators on them. For example, in the sentence *It is clear that toxic doses of digitalis are regularly associated with a loss of myocardial K*, a simple version of this analysis is shown by grouping the words of the sentence into levels corresponding to S_e and the successive operators:

toxic doses of
|
It is clear that (digitalis is regularly associated with (a loss of myocardial K)).

When sentences from articles in the science subfield are decomposed by any one of the above methods, it is found that the vocabulary which is characteristic of the subfield (called here the science-specific vocabulary) occurs in a distinguished portion of the decomposition, i.e., in nodes corresponding to S_e and the immediate operators on S_e (the "bottom" nodes of the lattice or string decomposition), while the more general science vocabulary is at the intermediate nodes of the lattice or string decomposition. The top nodes are occupied by epistemic vocabulary presenting the scientist's relation to the science facts.*

Form of S_e

When we consider the science-specific verbs in the bottom-most nodes of the sentence decomposition, i.e., the verbs in S_e, we find that the subject of these verbs is a science-specific noun, and the object (if the verb is transitive) is also a science-specific noun, or several, interspersed with prepositions (e.g., *the cell loses potassium, ions flow into the cell*). Letting N and V stand respectively for the science-specific nouns and verbs in S_e, and P for a preposition selected by the given verb, a formula for the elementary sentence is:

$$S_e = N_1VP_1N_2P_2N_3$$

where a given verb may have only a portion of the $P_1N_2P_2N_3$ sequence as its object, or in some cases a longer sequence.

In the sublanguage many of the science-specific verbs have only one or two object possibilities, fewer than in their use in English as a whole. In some cases a prepositional phrase would be an object of a verb in the sublanguage whereas in English as a whole it would be considered an adjunct, e.g., *exchange* (*across membrane*). This fact reduces the ambiguity in the sentence analysis, and simplifies the work of obtaining a sentence analysis by computer.

N sets in S_e

A compact description of the main types of elementary sentences is obtained if we collect the science-specific nouns into (almost entirely) disjoint sets, chief of which are:

G (pharmacological e.g., glycosides, digitalis,

* A monograph presenting the complete grammar and the empirical methods used to obtain it is in preparation.

* In the string parses obtained by the computer, this separation of vocabulary appears clearly, since in the lexicon available to the parser, words are classed only with regard to their syntactic properties for English as a whole (noun, verb, etc.). Yet the science-specific vocabulary is found consistently in the lowest string-nodes of the decomposition, and the vocabulary expressing the scientist's conclusions, doubts, speculations, etc., is found in the highest nodes.[4]

agent)	digoxin, ouabain, erythrophleum alkaloids
I (ion)	e.g., K^+, Na^+, Ca^{++}, potassium, sodium, calcium
T (tissue)	e.g., muscle, strips of ventricle, vesicles, epithelium, fibers
C (cell)	e.g., cell, red cell
M (membrane)	e.g., membrane
H (heart)	e.g., heart, atrium, myocardium
O (other organs)	e.g., kidney
F (fluid)	e.g., fluid, medium solution, suspension

Certain nouns in these sets are pure synonyms in the sublanguage, completely interchangeable under whatever verb they occur with: *sodium*, *sodium ions*, *Na* (the first two are of course not synonyms in other areas of science writing).

Certain words are classifiers of particular sets (e.g., *ion* for K, Na, Ca, Cl), with such word-sequences as *these ions* being synonyms for particular ones of these in a particular textual occurrence. There are also verbs which are used as classifiers of certain sets of verbs (e.g., *act*).

Certain nouns occur as fragment-names of other nouns. A noun N_1 occurs as a fragment-name of N_2 if there exists a possible sublanguage sentence "N_1 is a part of N_2," and if in the given occurrence, N_1 occurs as the subject/object of a verb which elsewhere has N_2 as its subject/object. For example, in *The glycoside inhibits the Michaelis component of influx, the Michaelis component* is a fragment-name of *influx.*

In considering the combinations of nouns and verbs occurring in the texts, we note that while each of the above noun sets appears uniquely as the subject or object of certain verbs, there are also verbs which take their subject or object from particular unions (marked /) of these sets. There are also verbs which take their subject or object only from particular subsets of these sets (e.g., only sodium and potassium in I, or only Ca).

V-sets of S_e, and main S_e subtypes

Verb subclasses can be set up on the basis of verb occurrences in particular environments composed of the above noun sets. The environments are cases of the S_e formula. Some of the main environments for classing S_e verbs are listed below, followed by a sample list of verbs in each class. The statement of the subject and object noun classes with which a given verb on the list occurs is limited to the occurrences of that verb in the sentences of the articles which were analyzed. The verb classes are largely disjoint, but a given verb may be in more than one class. Verbs whose active form would have a human subject and a science-specific noun as object are stated in the passive form.

T/H__: contract, relax; is isolated.
T/C__: is washed, is cooled, is cold-stored, is warmed, is incubated, is fresh.
T__: is fractionated, is prepared, shortens.
C__: rest, swell, recover.
H__: beats, fails, is quiescent, is stimulated, survives, responds inotropically, functions, works, (has) activity.
M__: is permeable, is leaky. (These could be obtained from I__M, below).
I__I: replace, exchange with (across membrane).
I__C/T: move (in)to, enter, flow in/out, occupy (site in), is stored in, is sequestered in, concentrate in, accumulate in/at, distribute in, constitute composition of.
I/G__C/T: diffuse into, are in, leave, localizes in, is removed from.
I__M: permeate.
G__M: penetrate.
C__I: regain, expel, is loaded with.
C/T__I: extrude, eliminate, is depleted of, leaks, are deprived of, gain.
H/C/T__I/G: lose, take up.
O/T__I: excretes, turn over, release.
G__T: is absorbed into, is located in (region), reaches, combines with, is injected into; poisons, inactivates.
T__G: gets rid of, responds to, resists, is exposed to, is treated with.
F__F: equilibrate.
T__F: is suspended in, is surrounded by, is bathed in.
X__X (for any set X): is (ouabain is a glycoside).

Grouping the main S_e subtypes

If we consider the above list we note that there are only a few types of subject/object pairs for these verbs. To obtain a more compact representation, we define an inclusive tissue class $\bar{T} = T/C/H/M/O$, and an inclusive class $\bar{I} = I/G$. In terms of these super classes the main environments above can be summarized as follows, defining the verb classes V_T, V_{II}, V_{IT}:

$$\bar{T}V_T$$

$$\bar{I}V_{II}\bar{I}$$

$$\bar{I}V_{IT}\bar{T}$$

The additional type $\bar{T}V_{TI}\bar{I}$ can be included in the above types by taking the verbs in the passive.

While the grouping of S_e subtypes into supertypes is a convenient reduction of a large amount of data, the individual subtypes within one supertype may behave differently under further operators. This is the case with $\bar{I}V_{IT}\bar{T}$ where $IV_{IT}C$ (*ions leave cell*) occurs under such operator sequences as *digitalis inhibits* (see below) whereas $GV_{IT}C$ does not.

It is found that the verb classes defined in this way are very nearly disjoint. The noun super-classes above are disjoint collections of the virtually disjoint noun subclasses established above.

Furthermore, if we consider the verbs in the list, we find that with the exception of those noted below, most of the verbs refer to movement or the result of movement: moving in or through (*flow into*, *transport*), staying in place (*occupy*, *sequester*), being in a place by virtue of having moved (*concentrate*, *accumulate*, *distribute*), favor moving or staying (*select*, *resist*). Many of these verbs are indeed synonymous in respect to these elementary sentences, and the others could all be replaced in these elementary sentences by synonymous word sequences, a base verb *move* with particular prepositions and quantifiers (e.g., *permeate*: move through; *gain*: move in to a greater degree than move out, etc.). The verbs which do not relate to movement are mainly the intransitive and laboratory verbs at the beginning of the list, and certain particular verbs, such as *poison*, *inactivate*, *destroy* and *respond to* and *equilibrate* in the latter part of the list. This main set of elementary sentences of the subfield is thus composed of a single verb *move* with directional and quantitative modifiers which connect $\bar{I}$ to $\bar{T}$, (and $\bar{I}$ to $\bar{I}$ in respect to $\bar{T}$, e.g. *exchange*).

Other S_e subtypes

In addition to the main S_e subtype (covering ion transport phenomena) which is described in some detail above, there are several further S_e subtypes which are important in the subfield:

- S_e whose main nouns are *contractile proteins*, *actin*, *myosin* and characteristic verbs are *slide along*, *fold along* (*the sliding and perhaps folding of actin molecules*).
- S_e whose main nouns are *ATPase*, *ATP*. One such S_e has *ATPase* as subject and *ATP* as object, with *hydrolyze* as a characteristic verb. Most frequently the S_e verb occurring with *ATPase* is *act*, which is a classifier verb for more specific S_e verbs.
- S_e whose characteristic verbs are *carry*, *transport* (across membrane) with I (e.g., *sodium*) as characteristic object, and with *mechanism*, *substance*, *pump*, as frequent subjects, when the subject is given explicitly.

In addition to the above, in some articles or parts of articles, there are elementary sentences whose vocabulary is drawn (in part) from noun classes not mentioned above. Examples of these are: *The curve flattens toward the x-axis*, *cardiac glycosides possess unsaturated rings*, *the potential is negative*. These elementary sentences are found to be sentences of other, related, sciences and techniques on which our particular subscience draws.

Local modifiers of N and V; and wh-connectives

Certain additional words operate on the words of the S_e sentences. The operators on the nouns may appear as adjectives, prepositional phrases and other modifiers. The operators on the verb may be adverbs, prepositional phrases and other modifiers. The noun modifiers can be reconstructed into separate sentences connected by a relative pronoun (*that*, *which*, etc., indicated by *wh*) to the given sentence, and the verb modifiers into separate sentences connected to the given sentence by a bisentential verb V_{ss}. Below, in proposing a format for the content of each sentence, we will suggest that instead of transforming all modifiers out of the sentence, as one does for language as a whole, we consider if there are any word sets in modifier position which in this sublanguage are especially dependent on their host words, or which never have an explicit conjunctional relation to it; these, we suggest, might best be left in modifier slots next to their host word in the format.

Aspectuals, V_v

Certain verbs V_v (not science-specific verbs treated above) operate on verbs as more or less aspectual modifiers. In English, they occur either in pre-verb or post-sentence position, and most can be transformed from one to the other: *He commenced speaking*, *His speaking commenced*. In this sublanguage, only a few are used, and all are aspectual in meaning (including the negative), and apparently all can occupy the pre-verb position: *not*, *fail to*, *appear to*, *tend to*, *be engaged in*, *undergo*, *persist*, *continue*, *remain*, *become*, *commence*, *start*. E.g., *the force starts to increase*, *the steroids undergo interconversion*, *depolarization persists* (persist in depolarizing). Several of these are synonyms of each other in the sublanguage.

Quantifiers, Q

Certain verbs (e.g., *flow*, *transport*, *lose*, *gain*, *accumulate*) can have a modifying quantifier Q: *in an amount*, *at a rate*; or when the verb is nominalized: *amount of*, *rate of*. This holds for certain adjectives and nominalized adjectives (e.g., *toxicity*, *activity*) and even nouns (*force*). Some can even be considered to contain a quantifier (e.g., *concentration* is synonymous in this sublanguage with *amount of concentration*). Quantifiers can also be considered to be modifiers or predicates of certain nouns: *amount of digitalis, digitalis is present in a certain amount.*

There are certain other verbs (different from any listed in preceding sections) which operate on these Q. Of these, there is a subset V_q whose members have Q as their subject, and there is a subset V_{qq} whose members have Q as their subject and Q as their object. An example of V_q is *decrease* in *the size of the overshoot decreases*; an example of V_{qq} is *equals* in *the amount of alcohol in . . . , equals the amount of alcohol in . . . ,* and *the chloride ratio equals the potassium ratio.* A quantifier Q occurring with V_q or V_{qq} is often omitted (zeroed), since its original presence can be reconstructed from the grammatical requirements of the V_q or V_{qq}. Thus, in addition to: *raise the internal sodium concentration*, we find also: *raise the internal sodium.*

The chief verbs here are:

V_q: *decrease, reduce, fall, increase, rise, change, run down, level off, stand still.*

V_{qq}: *equal, differ from, range from__to__, be twice, vary with, correspond to, depend on, determine, reach.* Certain V_{qq} appear also with a human subject with the two Q's in the object: *compare, correlate* (an amount with an amount); *determine, calculate* (an amount from an amount).

There is also a $V_{V_qV_q}$, i.e., a verb having V_q both as subject and as object: *parallel* (*the increase in tension parallels the increase in uptake*). That a verb should require such a hierarchy of object-types is unique in the sublanguage, and not common, if it exists at all, in the language as a whole.

The V_q and V_{qq} can operate not only on Q but also on V_q: *the rise* (*in amount*) *depends on . . . ,* where *depend on* is a V_{qq} operating on *rise* and *rise* is a V_q operating on Q *amount.* There are also purely causative verbs whose objects are Q or V_q: *double, accelerate, minimize, depress.*

We see that a complex structure of quantifiers and quantifying verbs operates in this sublanguage. As in the case of the verbs reducible to *move*, above, many of the quantifying verbs here are synonyms, or are replaceable by a few base verbs with modifiers on them.

Verbs connecting two sentences, V_{ss}

There are certain verbs, not included in any of the preceding sets, which have nominalized sentences both as subject and object. These verbs are the bisentential V_{ss}. A particular property of these verbs is that if their first nominalized sentence is *presence of X* or *action of X*, where X = the noun subclass G (rarely, I), the words *presence of* or *action of* are omittable, yielding X as the apparent subject of the V_{ss}: *glycosides inhibit . . . .* These V_{ss} are: *affect, is concerned in, bring about, cause, produce, confer, make, generate, induce, initiate, trigger, promote, stimulate, prolong, protect from, restore, control, interfere with, inhibit, limit, delay, antagonize, depose, reverse, block, arrest, abolish, obstruct, prevent, switch off.*

Instead of considering *glycosides inhibit sodium efflux* as reduced from *action of glycosides inhibits sodium efflux* (an $SV_{ss}S$ construction), we can consider the G noun, when it appears as subject of V_{ss}, to constitute a special N-class N_0. Then *glycosides inhibit sodium efflux* would be a case of an $N_0V_{ss}S$ construction. We use the latter analysis in the format, below. Here, too, it is clear that there are many synonyms with respect to the use of these verbs in the sublanguage, so that the vocabulary could be reduced.

There are a few other sentence-connecting verbs which may be called conjunctional V_{ss}. Here, G does not occur as possible subject: *involve, accompany, relate to, lead to, depend on, be based on.* Similar to these are certain passive forms: *be linked to, be coupled to, be related to,* which in the active form have a human subject.

Subordinate conjunctions, C_{subord}

There are a number of subordinate conjunctions between sentences: *if S then S, S when S*, etc. There are also certain prepositions used conjunctionally between nominalized sentences: *No contracture occurs on depolarization, Recovery does not occur in the absence of oxygen.*

Coordinate conjunctions, C_{coord}

There are conjunctions between S, or between identically classed words: *and, or.*

Sentence grouping (*non-associativity of connectives*)

All the sentence connectors, including *wh*, can operate on each other, i.e., an $SV_{ss}S$ or an SCS can serve as subject or object of a sentence connector. When there is more than one connective, the grouping of sentences is semantically non-associative, but sequences of SCS, where $C = C_{coord}$, are associative.

Epistemic operators

Finally, there are many verbs with epistemic meaning, whose subject is human and whose object is a sentence: *believe, publish.* The human subject is often omitted when the sentence is nominalized in the passive.

Summary

Grammar. This sublanguage had a definite grammatical structure consisting of:

(1) a set of elementary sentences, formed out of a few sets of subscience-specific nouns and verbs; and occasional other elementary sentences of a few other subscience vocabularies.
(2) aspectual operators on verbs.
(3) (omittable) quantifiers Q on certain verbs or nouns, with quantifying verbs V_q and V_{qq} operating on the Q or on V_q; and a verb V_{VqVq} operating on two V_q's.
(4) the noun-modifying *wh*-connective.
(5) sets of sentence-connecting verbs V_{ss} and conjunctions C, which can operate on each other.

Vocabulary reduction. In each word set, various words are used synonymously or can be replaced by a common base word with differentiating modifiers. Hence the vocabulary in each word set can be greatly reduced, at least for the purposes of a standardized informational representation.

Semantic interpretation. The particular word sets (especially after their vocabulary has been reduced) and the way they operate on each other reflect quite closely the structure of information in the science. E.g., a main S_e subtype is *I/G move in T/C*; and the main appearance of glycosides is not in the elementary sentence, but as subject of the causative operator verb on the S_e. Also, the complexity of the quantity words reflects the importance of quantitative relations in this subfield.

SUBLANGUAGE SENTENCE FORMAT

A sublanguage grammar provides a basis for structuring the information in each sentence and for mechanically processing the structured information.

A parse of a sentence, whether carried out by hand or by a computer program, is a decomposition of the sentence into parts which are segmented, and related one to the other, in terms of the grammar used. When the grammar includes, in addition to the grammatical requirements and transformations of the language as a whole, also the special word subsets and restricted combinations of the given science sublanguage, the sentence segments and their relations are found to fit the informational categories and relations of the subscience. It is possible to construct a fixed format of the grammatical operators and operands which houses all the sentence outputs obtained using the sublanguage grammar, so that the grammatical decomposition (parse) of each sentence locates the sentence-segments in particular slots of the format. Each of the slots has a fixed informational character, and each sentence carries the type of information of the slots which it fills, in their relation to neighboring slots in the format.

Aside from the sublanguage grammar, it is known that in language in general there are certain grammatical processes which lead to the loss of words in a sentence or to the replacement of words by informationally less explicit ones. The reverse process of supplying the lost or more specific words is especially important in formatting sentences. The main such processes are:

(1) Loss of repeated words (called "zeroing"), especially after a conjunction. E.g., *changes in the concentration of electrolytes and in electrolyte fluxes* can be filled out to include the zeroed word *changes* after *and,* to yield *changes in the concentration of electrolytes and* [*changes*] *in electrolyte fluxes.* In the formatted sentences, below, zeroed words which have been reconstructed are enclosed in [ ].
(2) Replacement of a repeated word or sentence by a pronoun, e.g., *its* in *the inotropic action of digitalis cannot be attributed to its effect on potassium metabolism,* and *This* in *This results from a slowing of the influx.* A so-called bound pronoun occurs in words like *which,* which can be analyzed as a conjunction *wh* followed by a pronoun *ich* standing either for a preceding noun or sentence. In the formatted sentence, material which has been reconstructed in place of a pronoun is enclosed in { }.
(3) Replacement of a repeated word or sentence by a classifier of the word or sentence, usually as part of a sequence containing *the, this, these,* etc., e.g., *the drug* replacing a second occurrence of *digitalis* in the same sentence, or *these effects* replacing the repetition of a preceding sentence. The combination of a pronominal element (e.g., *these*) with a classifier word or phrase eases the task of identifying the antecedent of the pronoun. In the formatted sentences, material which

has been reconstructed on the basis of classifier sequences is enclosed in ⟨ ⟩.

(4) Grammatical constants. When a sentence occurs as the subject or object of an operator verb, the sentence may be nominalized, e.g., *an influx of potassium into the cell* following the operator verb *results from,* nominalized from *potassium flows into the cell.* In reconstructing the sentence which had been nominalized it is sometimes necessary to supply an informationally neutral word in order to make the nominalized form of the verb into a grammatical verb form. E.g., *intracellular sodium* in *Intracellular sodium is increased,* can be reconstructed into a sentence *sodium (is) intracellular.* In the formatted sentences, grammatical words which are supplied are enclosed in ( ).

U_1	C_1
U_2	C_2
	C_{N-1}
U_N	.

Figure 1—Sentence Format
U = Unary Sentence; C = Conjunction

The structure of a sentence in this grammar, after lost and replaced material has been reconstructed as far as possible, consists of a unary sentence U, or a sequence of U's connected by conjunctions C, where $C = V_{ss}$, V_{qq}, V_{VqVq}, C_{subord}, C_{coord}, *wh,* defined in the grammar above. This sequence, UCUC . . . CU, is the resultant of each C operating on a pair of sentences, where each of these operand sentences is itself a U or the resultant of a C operating on a pair of sentences. If there is more than one C in a sentence, parentheses are needed to indicate the hierarchical (non-associative) grouping of the operands of C's. In a formatted sentence, the U's and C's are arranged in columns, as in Figure 1. Grouping is shown by barred square brackets ⁅ ⁆. If no grouping is shown, it is understood that each C operates on the last preceding U.

N_0 V_{ss}	V_q	S_e	D_s

Figure 2—Format for Unary Sentence
N_o: subject of V_{ss}, generally a pharmacological agent noun
V_{ss}: operator on V_q or S_e, generally causative verb
V_q: verb of quantity
S_e: elementary sentence of the sublanguage
D_s: adverb on the whole preceding structure to its left
The first two boxes and the last may be empty; the first and last boxes are repeatable

Each unary sentence U is an elementary sentence S_e, or an S_e with one or more unary operators on it. The unary operators are either N_0V_{ss} or V_q, as illustrated in Figure 2. Each S_e is one of the S_e subtypes described in the grammar above. The gross grammatical structure of S_e is illustrated in Figure 3. In all the above formats, certain particular word subsets which appear in the structure may have particular sets of local modifiers operating on them. The grammatical form of these modifiers on nouns, is: adjectival phrases and possibly quantifiers Q; and on verbs: aspectual per-verbs V_v, adverbial phrases D and quantifiers Q.

Tables I-III contain the formats for the first two paragraphs of the section *Effects on cellular potassium* in a review of Digitalis.[8] In the textual sentence preceding each format, the "scientist-level" portions are italicized; these can be separated grammatically from the more specifically science portions of the sentence, and have not been included in the formats. The format follows the pattern illustrated in Figures 1-3. The elementary sentence S_e appears between double lines, and where the verb V occurs in the sentence in nominalized form, that form has been retained in the format. The first preposition following a verb in S_e has been written along with the verb in the V column.

With regard to the individual sentences:

N_1 Q	V Q	P N_2	P N_3

Figure 3—Format for Elementary Sentence S_e
N_1, N_2, N_3: Nouns from the specifically pharmacological vocabulary
Q: Quantity word
V: Verb or word with verb root
Only N_1 and V are necessarily present in each S_e

TABLE I—Formatted Sentences 1-3CONJ: conjunction; G *pharmacological agent* noun class; C: *cell* noun class; V_{IT}: verb with subject noun from $\bar{I}$ (*ion* super-class) and object noun from $\bar{T}$ (*tissue* super-class); other symbols are noted in Figure 1-3

1. CHANGES IN THE INTERNAL MILIEU OF CELLS PRODUCED BY DIGITALIS HAVE BEEN KNOWN FOR MANY YEARS.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
1.)	G Digitalis		produces	changes		←------internal milieu of-----→			C cells		

2. MOST PROMINENT HAVE BEEN CHANGES IN THE CONCENTRATION OF ELECTROLYTES AND IN ELECTROLYTE FLUXES, THAT IS, IN THE RATE OF MOVEMENT OF ELECTROLYTES IN AND OUT OF THE CELL.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
2.1)	←------------						------<[1]>----			------→	(has)most prominent (case in)
2.2)	G [Digitalis]		[produces]	changes in		I electrolytes	(have)concentration [in]		C [cells]		and
2.3)	G [Digitalis]		[produces]	[changes] in		I electrolytes	fluxes P		C [cells]		,that is,
2.4)	G [Digitalis]		[produces]	[changes] in		I electrolytes	movement in (at a) rate		C the cell		and
2.5)	G [Digitalis]		[produces]	[changes in]		I [electrolytes]	[movement] out of [(at a) rate]		C [the cell]		

3. INTEREST WAS INITIALLY FOCUSED ON CHANGES IN POTASSIUM; MORE RECENTLY, CHANGES IN CALCIUM HAVE BEEN RECOGNIZED TO BE OF GREAT IMPORTANCE.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
3.1)				changes in		I potassium	V_{IT}		$\bar{T}$		;
3.2)				changes in		I calcium	V_{IT}		$\bar{T}$		

TABLE II—Formatted Sentences 4-6

Symbols as in Table 1

4. TOXIC DOSES OF DIGITALIS CONSISTENTLY REDUCE THE INTRACELLULAR CONCENTRATION OF POTASSIUM IN A WIDE VARIETY OF CELLS, INCLUDING CARDIAC MUSCLE CELLS.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
4.1)	G Digitalis	toxic doses		reduces consistently		I potassium	(has)concentration intra-		C cellular	in a wide variety of cells	including
4.2)	G [Digitalis]	[toxic doses]		[reduces consistently]		I [potassium]	(has)[concentration intra-]		C [cellular]	in cardiac muscle cells	.

5. THIS RESULTS FROM THE SLOWING OF THE INFLUX OF POTASSIUM INTO THE CELL.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
5.1)	←------------						---- {4} --------			------→	results from
5.2)				slows		I potassium	influx into		C the cell		. Concurrently

6. CONCURRENTLY, INTRACELLULAR SODIUM AND WATER ARE INCREASED.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
6.1)				increases		I sodium	(is) intra-		C cellular		and[concurrently]
6.2)				[increases]		I water	[is intra-		cellular]		

TABLE III—Formatted Sentence 7

Symbols as in Table 1

7. IT IS NOT CERTAIN WHETHER THESE LINKED CHANGES IN SODIUM AND POTASSIUM ARE PRODUCED BY A SINGLE EFFECT OR ARE SEPARATELY MEDIATED.

	N_0	Q	V_{ss}	V_q	D	N_1	V	Q	N_2	D_s	CONJ
7.1)	single effect		produces [	←--------	------	----------	-----<6.1>	-------	--------	---------→	and
7.2)				←--------	------	----------	-----<5.2>	-------	--------	---------→	ǂ or
7.3)	ǂN		mediates separately	←--------	------	----------	-----[6.1]	-------	--------	---------→	[and]
7.4)	N		[mediates separately]	←--------	------	----------	-----{5.2]	-------	--------	---------→	ǂ wh
7.5)				←--------	------	----------	-----{<6.1>}	-------	--------	---------→	is linked to
7.6)				←--------	------	----------	-----{<5.2>}	-------	--------	---------→	.

In (1.): While *changes in cells produced by digitalis* is ambiguous in English as a whole, it is not ambiguous in the sublanguage since nouns in the class C (*cells*) do not occur as the subject of V_{ss} verbs (*produce*). In the sublanguage the word *changes* only operates on quantity words Q (e.g., *amount, rate*) or verbs which have an implicit Q on them. In the formats, therefore, *change* occupies the V_q position. In the format for sentence 1 this places *internal milieu of cells* in the S_e position, suggesting that it contains an implicit Q and V. This is supported by the fact that the paraphrase *changes in the amounts of* X_1, X_2, . . . *in cells* is an acceptable substitute for *internal milieu of cells* in all its textual occurrences in this sublanguage.

In (2.): The first part of sentence 2 contains lost repeated material (zeroing) which can be reconstructed because of the strong grammatical requirements on the superlative form: *Most prominent have been changes in* . . . is filled out to *Most prominent of these changes have been changes in* . . . . *These changes* is a classifier sequence replacing the full repetition of sentence 1, which is then shown in the format as the first (zeroed) unary sentence of 2.

In (2.3-2.5): The word which indicates that *changes* (along with *digitalis produces*) has been zeroed is the repeated *in* after *and.* In 2.2, the V in S_e is (*have*) *concentration in* (or: *concentrate to some amount in*), which in the sublanguage requires an object noun from the gross tissue-cell class $\bar{T}$. Similarly in 2.3, the V *fluxes* (with unspecified P) requires an object noun from $\bar{T}$. In the analyzed texts both of these Vs occurred almost exclusively with the noun *cell* as their object. The definitional connective *that is* between 2.2,3 and 2.4,5 supports substituting the word *cell* for $\bar{T}$.

In (3.): The sublanguage requirements on the noun class I (*potassium, sodium*) as the first noun in S_e, when S_e is operated on by V_q (*changes*), are that the verb be of the type V_{IT} or V_{II} and the second noun be of class $\bar{T}$ or $\bar{I}$. The continuity of this sentence with its surrounding sentences suggests that the verb is V_{IT} and the noun $\bar{T}$ (more specifically C:*cell*).

In (5.1): The pronoun *this* replaces the entire preceding sentence.

In (7.): *These linked changes in sodium and potassium* transforms into *These changes in sodium and potassium which are linked.* The portion up to *which are linked* is a classifier of the two preceding unary sentences, 6.1 and 5.2, pinpointed by the repetition of the words *sodium* and *potassium* in the classifier sequence. It is these two conjoined unary sentences which are operated on by *a single effect produces* in lines 7.1, 7.2, and again by *mediates separately* with unknown N subject, in lines 7.3, 7.4. The portion *which are linked* applies to both occurrences of 6.1 and 5.2 in 7.1-4. The *wh* in *which* is the connective and the *ich* part is a pronoun for the two sentences, as indicated by { }. The fact that the sentences were reconstructed by use of a classifier is indicated by the ⟨ ⟩ inside the { } in 7.5-6. Although this sentence seems empty, it is common in scientific writing for a sentence to consist of references to previous sentences with new operators and conjunctions operating on the pronouned sentences. The linearity of language makes it difficult to express complex interconnections between the events (sentences) except with the aid of such pronouned repetitions of the sentences.

The appearance of a word like *effect* in the column usually filled by a pharmacological agent noun G may herald the future occurrence of a new elementary sentence or a new set of conjoined elementary sentences (classified by the word *effect*) which will intervene between G and the present S_e. This appears to be one of the ways that new knowledge entering the subfield literature is reflected in the formats and the sublanguage grammar.

In fact, in the work described here, the first investigation, which covered digitalis articles up to about 1965, showed certain sets of words (including *mechanism*, *pump* and, differently, *ATPase*) appearing in the N_0 or D_s column as an operator on S_e. In later articles, which were investigated later, these nouns appeared increasingly as subjects of new S_e subtypes listed above in the grammar, connected by conjunctions to the previously known S_e. The shift of these words from occurring as operators to occurring in (or as classifiers of) new S_e subtypes is the sublanguage representation of the advance of knowledge in the subfield.

ACKNOWLEDGMENTS

This work was supported by Research Grants R01 LM 00720-01, -02, from the National Library of Medicine, National Institutes of Health, DHEW. Important parts of the sublanguage grammar are the work of James Munz, to whom many of the results and methods are due.

REFERENCES

1 F W LANCASTER
Evaluation of the Medlars demand search
National Library of Medicine 1968

2 Proceedings of 1971 Annual Conference of the ACM pp 564-577

3 N SAGER
Syntactic analysis of natural language
Advances in Computers 8 F Alt and M Rubinoff eds Academic Press New York 1967

4 N SAGER
The string parser for scientific literature
Courant Computer Symposium 8—Natural Language Processing R Rustin Ed Prentic Hall Inc Englewood Cliffs N J In press

5 String Program Reports Nos 1-5
Linguistic String Project New York University 1966-1969

6 D HIZ A K JOSHI
Transformational decomposition—A simple description of an algorithm for transformational analysis of English sentences
2eme Conference sur le Traitement Automatique des Langues Grenoble 1967

7 String Program Reports No 6
Linguistic String Project New York University 1970

8 A F LYON A C DEGRAFF
Reappraisal of digitalis, Part I, Digitalis action at the cellular level
Am Heart J 72 4 pp 414-418 1961

Dimensions of text processing*

by GARY R. MARTINS

University of California
Los Angeles, California

INTRODUCTION

Numerical data processing has dominated the computing industry from its earliest days, when computing might better have been called a craft than an industry. In those early days it was not uncommon for a mixed group of scientists and technicians to spend an entire day persuading a roomful of vacuum tubes and mechanical relays to yield up a few thousand elementary operations on numbers. The emphasis on numerical applications was a wholly natural consequence of the dominant interests of the men and women who designed, built, and operated those early computing machines.

Within a single generation, things have changed dramatically. Computing machines are vastly more powerful and reliable, and easier to use thanks to the efforts of the software industry. Perhaps of equal importance, access to computers can now be taken for granted in the more prestigious centers of education, commerce, and government. And we may be approaching the day when computing services will be as widely available as the telephone. But it is still true that numerical data processing—"number crunching," in one form or another—is the principal application for computers.

That, too, is changing, however. Due principally, I think, to the highly diversified needs and interests of the greatly expanded community of computer users, the processing of textual or lexicographic materials already consumes a significant percentage of this country's computing resources, and that share is rising steadily. By the end of this decade, if not before, text processing of various kinds may well become the main application of computers in the United States.

This prediction, no doubt, carries the ring of authentic good news for those of us with strong interests in one or another of the many kinds of textual data processing. But we must face the fact now that there remains a serious and large-scale educational task that must be undertaken if the future growth of textual data processing is to fulfill the high hopes for it that we now entertain. Text processing tasks and systems are too often considered in isolation from one another, with the results that (1) much design and implementation work needlessly duplicates prior accomplishments, and (2) potentially useful generalizations and extensions of existing systems for new applications are overlooked.

This is a tutorial paper, then. My purpose is to take a broad view of the text processing field in such a way as to emphasize *the relations among different systems and applications*. The structure of these relationships will be embedded in an informal descriptive space of two dimensions. In the interests of focussing attention on the unifying character of this framework, however imperfect and incomplete it surely is, I shall avoid the discussion of the internal details of specific systems.

TEXTUAL DATA PROCESSING

By "textual data processing" I mean a computing process whose *input* consists entirely or substantially of character strings. For the most part, it will be convenient to assume that this textual input represents natural language expressions, such as, for example, sentences in English. All kinds of systems running today fit this deliberately broad and usefully loose definition: programs to automatically make concordances, compile KWIC indexes, translate between languages, evaluate personnel reports, drive linotype machines, abstract documents, answer questions, perform content analysis, route documents, search libraries, and edit manuscripts. I am sure everyone here could add to this list. It will be instructive to include programming language compilers in our discussion, as well.

* Research reported herein was conducted under Contract #F30602-70-C-0016 with the Advanced Research Projects Agency, Department of Defense.

TWO DIMENSIONS OF TEXT PROCESSING

An important set of relationships among these highly diverse activities can be clarified by locating them in an informal space of two dimensions: *scope* and *depth of structure*. The dimension of scope has to do with the magnitude of the task to be performed: the size of the problem domain, and the completeness of coverage of that domain. To illustrate, an operating production-oriented Russian-to-English machine translation system has a potentially vast input domain, namely, all the sentences of Russian. But an experimental model for such a system, containing only a tiny dictionary and a few illustrative grammar rules—something concocted for a demonstration, perhaps—has a highly restricted domain. The scope of the two systems differs greatly, with important consequences which we shall consider in a moment.

The second of our dimensions measures the richness of structure or "vertical integration" developed for the text. This is essentially a linguistic dimension, reflecting the fact that the text itself is made up of natural language expressions of one kind or another. This dimension does *not* take into account simple linear physical divisions of text, such as the "lines" and "pages" of text editing systems like TECO.[1] Rather, it measures an essentially non-linear hierarchy of abstract levels of structure which define the basic units of interest in a given application.

Scope

Generally, the dimension of scope as applied to the description of text processing systems does not differ in any systematic way from the notion of scope as applied to other systems. It enables us to express the relative magnitude of the problem domain in which the system can be of effective use. There are two key factors to be considered in estimating the scope of a particular system. The first of these has to do with the generality of input data acceptable to the system. If the acceptable input is heavily restricted, the scope of the system is relatively small. Voice-actuated input terminals have been rather vigorously promoted during the past few years; their scope is small indeed, being limited to the effective recognition of a very small set of spoken words (names of digits and perhaps a few control words), and often demanding a mutual "tuning" of the terminal and its operators. Programming language compilers provide another rather different example of systems with limited scope in terms of acceptable input; both the vocabulary and syntax of acceptable statements are rigidly and narrowly defined. In contrast, text editing systems in general have wide scope in terms of acceptable input.

The other factor which plays an important role in determining the scope of text processing (or other) systems has to do with the convenience and flexibility of the interface between the system and its users. Obviously, this factor will be of lesser importance in the evaluation of systems operated as a service in a closed-shop batch-processing environment. It will be of major importance in relation to systems with which users become directly involved, interactively or otherwise. Compilers are a good example of such systems, as are such widely-used statistical packages as SPSS[2] and BMD,[3] and interactive processors like BBN-LISP,[4] BASIC[5] and JOSS.[6] More to our present point are text-editing systems such as TECO, QED,[7] WYLBUR,[8] HYPERTEXT[9] and numerous others; in terms of acceptable data input, these latter systems impose few restrictions, but they may be said to differ significantly in overall scope on the basis of differences in their suitability for use by the data processing community at large. It takes a sophisticated programmer with extensive training to make use of TECO's powerful editing and filing capabilities, for example; this restricts the system's scope. A most ambitious assault on this aspect of the problem of scope in text editing systems is that of Douglas Englebart and his colleagues at Stanford University;[10] a review of their intensive and prolonged efforts should convince anyone of the serious nature of the difficulties involved in widening the general accessibility of text editing systems.

I am sure we have all had experiences with text processing systems of very restricted scope. A decade ago, it was a practice of some research organizations to arrange demonstrations of machine translation systems; in some memorably embarrassing instances, the scope of these systems was unequal to even the always quite carefully hedged, and sometimes entirely pre-arranged, test materials allowed as input. More commonly, we may have written or used text processing programs of one kind or another which were created in a deliberately "quick and dirty" fashion to answer some special need, highly localized in space and time. It is important to note that the highly restricted scope of such "one-shot" programs in no way diminishes their usefulness; given the circumstances, it may indeed have involved an extravagant waste of resources to needlessly expand their scope in either of the two ways I have mentioned.

While it may be quite difficult to measure the relative scope of different text processing systems, at least the basic notions involved are simple: breadth of acceptable input data and, where appropriate, the breadth of the audience of users to which the system is effectively addressed. Let us now review the *depth of structure* dimension of text processing, where the basic notions involved may be somewhat less familiar.

Depth of Structure

We may assume that the text to be processed first appears in the system as a continuous stream of characters. It will rarely be the case that our interest in the data will be satisfied at this primitive level of structure. We will most often be interested in words, or phrases, or sentences, or meanings, or some other set of constructs. Since these are not given explicitly in the character stream, it will be necessary to operate on the character stream to derive from it, or assign to it, the kinds of structures that will answer our purposes. The lowest level of structure, in this sense, consists of the sequence of characters in the stream. The highest level of structure might perhaps involve the derivation of the full range of meanings contained in and implied by the text. Between these extremes we may define a variety of useful and attainable structures. The dimension along which we measure these differences is that to which I have given the somewhat clumsy name of *depth of structure.*

The number of useful applications of text processing at the lowest level of structural depth—the character stream—is quite large. Most text editing systems operate at this level. Other more specialized applications include the development of character and string occurrence frequencies, of use principally to cryptanalysts and communications engineers. But, for applications which cannot be satisfied by simple mechanical pattern-seeking and matching operations, we must advance at least to the next level of structure, that of the pseudo-word.

Pseudo-words

The ordinary conventions of orthography and punctuation enable us to segment the character stream into word-like objects, and also into sentences, paragraphs, etc. The word-like objects, or pseudo-words, may be physically defined as character strings flanked by blank characters and themselves containing no blank characters. This is still a fairly primitive level of structure, and yet it suffices for many entirely respectable applications. Concordances have often been made by programs operating with this level of textual structure, for example. But there are serious limitations on the results that can be achieved. It is not possible, for instance, for the computer to determine that "talk" and "talked" and "talks" are simply variants of the same basic word, and that they should therefore be treated similarly for some purposes. The same difficulty appears in a more refractory form with the items "go" and "went." Thus, if our intended application will require the recognition of such lexicographic variants as members of the same word family, it will be necessary to approach the next more or less clearly defined level of textual structure, that of true words.

Word recognition

There are two principal tools used for the recognition of words in text processing: *morphological analysis* and *dictionaries.* They come in many varieties, large and small, expensive and cheap. Either may be used without the other. The Stanford Inquirer content-analysis system[11] employs a very crude kind of morphological analysis which consists in simply cutting certain character strings from the ends of pseudo-words, and treating the remainder as a true word. This procedure is probably better than nothing at all, but it can produce some bizarre confusions; for example, the letter "d" is cut from words ending in "ed," on the assumption that the original is the past tense or past participle form of a verb. "Bed" is thus truncated to "be," and so on. The widely used and highly successful KWIC[12] indexing system operates with a crude morphological analysis of essentially this kind.

More sophisticated morphological analysis is attainable through the use of more flexible criteria by which word-variants are to be recognized, at the cost of a correspondingly more complex programming task. But it is hard to imagine what sort of rules would be needed to cope with the so-called strong verbs of English. The best answer to this problem is the judicious use of a dictionary together with the morphological analysis procedures.[13] In systems employing both devices, a pseudo-word is typically looked up in the dictionary before any analysis procedures are applied to it. If the word is found, then no further analysis is required, and the basic form of the word can be read from the dictionary, perhaps together with other kinds of information. Thus, "bed" would be presumably found in the dictionary, preventing its procrustean transformation to "be." Likewise, "went" would be found as a separate dictionary entry, along with an indication that it is a variant of "go." On the other hand, "talked" would presumably not be found in the dictionary, and morphological analysis rules would be applied to it, yielding "talk"; this latter would be found in the dictionary, terminating the analysis.

Here I have tacitly assumed that our dictionary contains all the words of English, or enough of them to cover a very high percentage of the items encountered in our input text. In fact, there are very few such dictionaries in existence in machine-usable form. The reason is twofold: on the one hand, they are very expensive to create, and generally difficult to adapt, requiring a level of skills and time available only to the more heavily endowed projects; on the other hand, while they provide an elegant and versatile solution to the problems of word identification, most current text processing applications simply do not require the degree of versatility and power that large-scale dictionaries can provide.

A number of attempts have been made in the past to build automatic document indexing and dissemination systems based upon the observed frequencies of words in the texts. In these and similar systems, it was found to be necessary to exclude a number of very frequent "non-content-bearing" words of English from the frequency tabulations—such words as "the," "is," "he," "in," which we might collectively describe as members of closed syntactic classes: pronouns, demonstratives, articles, prepositions, along with a few other high frequency words of little interest for the given applications. The exclusion of these words is accomplished through the use of a "stop list," a mini-dictionary of irrelevant forms. Such small, highly specialized dictionaries are easily and cheaply constructed, and have proven useful in a wide variety of applications.

Automatic typesetting systems provide another good example of a useful text processing application operating at the level of the word on the dimension of depth of structure. The key problem for these systems is that of correctly hyphenating words to make possible the right and left justification of news columns. Morphological analysis of a quite special kind is employed to determine the points at which English words may be broken, and this analysis is often supplemented with small dictionaries of exceptional forms. Another more sophisticated but closely related application of rather narrow but compelling interest is that of automatically translating English text into Braille. Once again, specialized word analysis, supplemented by relatively small dictionaries of high frequency words and phrases, have been the tools brought to bear on the problem.

Before moving on to consider text processing applications of higher rank on the scale of depth of structure, I should like to pause for a moment to comment on what I believe to be a wide-spread fallacy concerning text processing in general. Somehow, a great many people, in and out of the field of text processing, have come to associate strong notions of prestigiousness exclusively with systems ranking at the higher end of the dimension of depth of structure. It is hard to know how or why this attitude has developed, unless it is simply a reflection of a more general fascination with the obscure and exotic. But it would be most unfortunate if capable and energetic people were for this reason diverted from attending to the many still unrealized possibilities in text processing on the levels we have been discussing. We have only to compare the widespread usefulness of text processing systems operating at the word level and below with the generally meagre practical contributions of systems located further along this dimension to dispel the idea that there is greater intrinsic merit in the latter systems. In now moving further along the dimension of depth of structure, we leave behind a broad spectrum of highly practical and useful systems that sort and edit text, make indexes, classify and disseminate documents, prepare concordances, set type, translate English into Braille, perform content analysis, make elementary psychiatric diagnoses from writing samples, assist in the evaluation of personnel and medical records, and routinely carry out many other valuable tasks. It may be only moderately unjust to repeat here a colleague's observation that, in contrast, the principal product of the systems we are about to consider has been doctoral dissertations.

Syntax

Syntax is, roughly speaking, the set of relations that obtain among the words of sentences. For some applications in the text processing field, syntactic information is simply indispensable. The difference between "man bites dog" and "dog bites man" is a syntactic difference; it is a difference of no account in applications based upon word frequencies, for example, but it becomes crucial when the functions or roles of the words, in addition to the words themselves, must be considered.

Syntactic analysis is most neatly accomplished when the objects of analysis have a structure which is rigidly determined in advance. The syntactic structure of valid ALGOL programs conforms without exception to a set of man-made rules of structure. The same is true of other modern programming languages, and of artificial languages generally. This fact, together with the tightly circumscribed vocabularies of such languages, makes possible the development of very efficient syntactic analyzers for them.

Natural languages are very different, even though some artificial languages, especially query and control languages, go to great lengths to disguise the difference. In processing ordinary natural language text we are confronted with expressions of immense syntactic complexity. And, while most artificial languages are deliberately constructed to avoid ambiguity, ordinary text is often highly ambiguous; indeed, ambiguity is a vital and productive device in human communication. The syntactic analysis of arbitrary natural language text is therefore difficult, expensive, and uncertain. It will come as no great surprise, then, that text processing systems that require some measure of syntactic analysis seldom carry the analysis further than is needed. Further, the designers of such systems have defined their requirements for syntactic analysis in a variety of ways. The result is that existing natural language text processing systems embody a great variety of analysis techniques. To some extent, this situation has been further complicated by debates among linguists as to what constitutes the correct analysis of a sentence, though the influence of these polemics has been minor. Over the past decade, and especially over the past five years or so, techniques for the automatic syntactic analysis of natural language text have improved rather dramatically, and are flexible enough today to accommodate a variety of linguistic hypotheses.

Earlier, in discussing the place of the dictionary in the identification of words, I mentioned that such dictionaries might carry other information in addition to the word's basic form. Often, this other information is syntactic, an indication of the kinds of roles the word is able to play in the formation of phrases and sentences. These "grammar codes," as they are often called, are analogous to the familiar "part of speech" categories we were taught in elementary school, though in modern computational grammars these distinct categories may be numerous. A given word may be assigned one or more grammar codes, depending upon whether or not it is intrinsically ambiguous. A word like "lucrative" is unambiguously an adjective. But "table" may be a noun or a verb. A word like "saw" exhibits even greater lexical ambiguity: it may be a noun or either of two verbs in different tenses.

The process of syntactic analysis, or parsing, generally begins by replacing the string of words—extracted from the original character stream as described earlier—by a corresponding string of sets of grammar codes. It then processes these materials one sentence at a time. Parsing in general does not cross sentence boundaries for the simple, though dismaying, reason that we know very little about the kinds of rule-determined connections between sentences, if indeed there are any of substance. On the other hand, the sentence is the smallest really satisfactory unit of syntactic analysis since we can be confident of our results for one part of a sentence only to the degree that we have successfully accounted for the rest of it, much as one is only sure of the solution to a really difficult crossword puzzle when the whole of it has been worked out.

If a sentence consists entirely of lexically unambiguous words—a rarity in English—then there is only a single string of grammar codes for the parser to consider. More commonly, the number of initially possible grammar code sequences is much higher; it is, in fact, equal to the product of the number of distinct grammar codes assigned to each word. Whatever the number, the parser must consider each of the possible sequences in turn, first assembling short sequences of codes into phrases—such as noun phrases or prepositional phrases—and then assembling the phrases into a unified sentential structure. At each step of the way, the parser is engaged in matching the constructs before it (i.e., word or phrase codes) against a set of hypotheses regarding valid assemblies of such constructs. The set of hypotheses is, in fact, the grammar which drives the parsing process. When a string of sub-assemblies corresponds to such a hypothesis (or grammar rule), it is assembled into a unit of the specified form, and itself becomes available for integration into a broader structure.

To illustrate, consider just the three words "on the table." The parser, first of all, sees not these words, but rather the corresponding string of grammar code sets: PREPOSITION ARTICLE NOUN/VERB.* Typically, it may first check to see whether it can combine the first two items. A table of rules tells the parser, as common sense tells us, that it cannot, since "on the" is not a valid English phrase. So, it considers the next pair of items; the ambiguity of the word "table" here requires two separate tests, one for ARTICLE + NOUN and the other for ARTICLE + VERB. The former is a valid combination, yielding a kind of NOUN-PHRASE ("the table"). The ARTICLE + VERB combination is discarded as invalid. Now the parser has before it the following: PREPOSITION NOUN-PHRASE. Checking its table of rules, it discovers that just this set of elements can be combined to form a PREPOSITIONAL-PHRASE, and the process ends—successfully.

This skeletal description of the parsing process is considerably oversimplified, and it omits altogether some important distinctive characteristics of parsing techniques which operate by forming broader structural hypotheses and thus play a more "aggressive" role in the analysis. The end result, if all goes well, is the same: an analysis of the input sentence, usually represented in the form of a labelled tree structure, which assigns to each word and to each phrase of the sentence a functional role. Having this sort of information, we are able to accurately describe the differences between "man bites dog" and "dog bites man" in terms of the different roles the words play in them. In this simple case, of course, the SUBJECT and OBJECT roles are differentially taken by the words "dog" and "man."

I remarked earlier that few production-oriented systems incorporate large-scale dictionaries. The same

* The notation "*X*/*Y*" is used here to indicate an item that may belong either to category *X* or to category *Y*.

is true of syntactic analysis programs; large-scale sentence analyzers are still mainly experimental. The parsing procedures of text processing systems that are widely used outside the laboratory, with a very few interesting exceptions, are designed to produce useful partial results of a kind just adequate to the overall system's requirements. Economies of design and implementation are usually advanced as the reasons for these limited approaches to syntactic analysis.

A meritorious example of limited syntactic analysis is provided by the latest version of the General Inquirer, probably the best known and most widely used of content-analysis systems. The General Inquirer-3,[14] as it is called, embodies routines which are capable of accurately disambiguating a high percentage of multiple-meaning words encountered in input text. This process is guided by disambiguation rules incorporated in the system's large-scale dictionary,[15,16] the *Harvard Fourth Psychosociological Dictionary.* These rules direct a limited analysis of the context in which a word appears, using lexical, syntactic, and semantically-derived cues to arrive at a decision on the intended sense of the word. In this manner, nine senses are distinguished for the word "charge," eight for "close," seven for "air," and so on.

There are language translating machine in daily use by various government agencies in this country. These machine translation systems are basically extensions of techniques developed at Georgetown University in the early 1960's. Their relatively primitive syntactic capabilities are principally aimed at the disambiguation of individual words and phrases, a task which they approach—in contrast with the General Inquirer-3—with an anachronistic lack of elegance, economy, or speed. For most purposes, the output of these systems passes through the hands of teams of highly-skilled bilingual editors who have substantial competence in the subject matter of the texts they repair. A most valuable characteristic of the government's machine translation systems is the set of very-large-scale machine-readable dictionaries developed for them over the course of the years. It is to be expected that major portions of these will prove to be adaptable to the more modern translation systems that will surely emerge in the years ahead.

A working system employing full-blown syntactic analysis is the *Lunar Sciences Natural Language Information System.*[17] This system accepts information requests from NASA scientists and engineers about such matters as the chemical composition and physical structure of the materials retrieved from the moon. The queries are expressed in ordinary English, of which the system handles a rich subset. Unrecognizable structures result in the user's being asked to rephrase his query. The requests are translated into an internal format which controls a search of the system's extensive data base of lunar information.

Semantics

The next milestone along the dimension of depth of structure is the level of semantics, which has to do with the *meanings* of expressions. Although semantic information of certain kinds has sometimes been used in support of lexical and syntactic processes (as, for example, in the disambiguation procedures of the General Inquirer-3), the number of working systems, experimental or otherwise, which process text systematically on the semantic level is close to zero. Those which do so impose strong restrictions on the scope of the materials which they will accept. The aforementioned lunar information system has perhaps the widest scope of any running system that operates consistently on the semantic level (with natural language input), and its semantics are closely constrained in terms of the concepts and relations it can process.

More flexible semantic processors have been constructed on paper, and a few of these have been implemented in limited experimental versions,[18,19,20,21] The more promising of these systems, such as RAND's MIND system,[22,23] are based upon the manipulation of semantic networks in which generality is sought through the representation of individuals, concepts, relations, rules of inference, and constructs of these as nodes in a great labelled, directed graph whose organization is at bottom linguistic. It is an unsolved problem whether such an approach can produce the needed flexibility and power while avoiding classical logical problems of representation.

The applications to which full-scale semantic processors might one day respond include language translation and question-answering or fact retrieval. At present, the often-encountered popular belief in super-intelligent machines, capable of engaging in intelligent discourse with men, is borne out only in the world of science fiction.

Beyond semantics

Pushing further along the dimension of depth of structure, beyond the rarified air of semantics, we approach an even more exotic and sparsely populated realm which I shall call *pragmatics.* There will be little agreement about the manner in which this zone should be delimited; I would suggest that systems operating at this level are endowed with a measure of operational self-awareness, taking an intelligent view of the tasks confronting them and of their own operations.

A remarkable example of such a system is Winograd's program[24] which simulates an intelligent robot that manipulates objects on a tabletop in response to ordinary English commands. The tabletop, the objects (some blocks and boxes), and the system's "hand" are all drawn on the screen of a video console. The objects are distinguished from one another by position, size, shape, and color. In response to a command such as

> "Put the large blue sphere in the red box at the right."

the "hand" is seen going through the motions necessary to accomplish this task, possibly including the preliminary removal of other objects from the indicated box. What is more, the system has a limited but impressive ability to discuss the reasons behind its behavior:

> Q: Why did you take the green cube out of the red box?
> A: So I could put the blue sphere in it.
> Q: Why did you do that?
> A: Because you told me to.

We should note that the scope of this system is in many ways the most restricted of all the systems we have mentioned, an observation which subtracts nothing from its great ingenuity.

SOME OBSERVATIONS

Now let me share with you a number of more or less unrelated observations concerning text processing systems which have been suggested by the two-dimensional view of the field which I have just outlined.

Information and depth of structure

We process text in order to extract in useful form (some of) the information contained in the text. In general, the higher a text processing system ranks on the dimension of depth of structure, the greater is the amount of information extracted from a given input text. This seems an intuitively acceptable notion, but I believe it can be given a more or less rigorous restatement in the terms of information theory. In sketching one approach to this end I shall attack the problem at its most vulnerable points, leaving its more difficult aspects as a challenge for others.

Consider a system which processes text strictly on a word-by-word basis, like older versions of the General Inquirer, for example. Such a system will produce identical results for all word-level permutations of the input text. But arbitrary permutations of text in general result in a serious degradation of information.† We conclude that text processing systems which operate exclusively at this level of depth of structure are intrinsically insensitive to a significant fraction of the total information content of the original text.

Similarly, syntactic processors whose operations are confined to single sentences (as is generally the case) are obviously insensitive to information which depends upon the relative order of sentences in the original text. And so on.

These considerations are of interest primarily because of their potential use in the development of a uniform metric for the dimension of depth of structure.

*The dimensions are continuous**

I want to suggest the proposition that *neither* of our dimensions is discrete, in the sense that it presents only a fixed number of disjoint positions across its range. That the *scope* of systems, in our terms, varies over a dense range is surely not a surprising idea. But the notion is fairly widespread, I believe, that the *depth of structure* of text processors can be described only in terms of a fixed number of discrete categories or "levels." Unfortunately, the use of the terms "syntactic level," "semantic level," etc. is difficult to avoid in discussing this subject matter, and it is perhaps the promiscuous use of this terminology which has contributed most to misunderstanding on this point.**

† It may not be easy to usefully quantify this information loss. The number of distinct word-level permutations of a text of n words is given by $n!/(f_1!\ f_2!\ \ldots\ f_m!)$ where f_j is the number of occurrences of the j-th most frequent word in a text with m distinct words. For word-level permutations, this denominator expression might be generalized on the basis of the laws of lexical distribution (assuming a natural language text), replacing the factorials with gamma function expressions. After that, one faces the thorny empirical questions: how many such permutations can be interpreted, wholly or in part, at higher levels of structure, and how "far" are these from the original?

* The term "continuous" is not meant to support its strict mathematical interpretation here.

** In the possibly temporary absence of a more satisfactory solution, a linear ordering for the dimension of depth of structure is derived informally in the following way. First, the various "levels" are mutually ordered in the traditional way (sublexical, lexical, syntactic, semantic, pragmatic, . . .) on the empirical grounds that substantial and systematic procedures on a given "level" are always accompanied by (preparatory) processing on the preceding "levels," but not vice-versa. Various theoretical arguments why this should be so can also be offered. Then, within a given "level" we may find various ways to rank systems with respect to the thoroughness of the job they do there.

As I have tried to indicate, there is in fact considerable variation among existing text processing systems in the degree to which they make use of information and procedures appropriate to these various levels. There are programs, for example, which tread very lightly into the syntactic area, making only occasional use of very narrowly circumscribed kinds of syntactic processes. Such programs are surely not to be lumped together with those which, like the MIND system's syntactic component,[25] carry out an exhaustive analysis of the sentential structure, merely because they both do some form of what we call syntax. The same is true of the other levels along the dimension of depth of structure; the great variety of actual implementations defies meaningful description in terms of a small number of utterly distinct categories. We use the terminology of "levels" because, in passing along this dimension of depth of structure we pass in turn a small number of milestones with familiar names; but it is a mistake to imagine that nothing exists between them.

Now I want to conjecture that it is precisely the quasi-continuous nature of this dimension which has helped to sabotage the efforts of researchers and system designers over the years to bring into being a small number of nicely finished text processing modules out of which everyone might construct the working system of his choice. The ambition to create such a set of universal text processing building blocks is a noble one and, like Esperanto, much can be said in its favor. But those who have worked on the realization of this scheme have not enjoyed success commensurate with the loftiness of their aims.

Why this unfortunate state of affairs? I believe it can be traced to the generally unfounded notion in the minds of text processing system designers, and their sponsors, that valuable economies of talent and time and money can be achieved by creating systems which, in effect, advance *as little as possible* along this dimension in order to get by. In fact, as is evident, for example, in some recent machine translation products, this attitude may be productive of needlessly complex and inflexible systems that are obsolescent upon delivery.

Of course, in many cases differences in hardware and software have prevented system designers from making use of existing programs. And in some cases the point might be made that considerations of operating efficiency dictated a "tight code" approach, ruling out the incorporation of what appears to be the unnecessary power and complexity of overdesigned components. But many of the systems in this field are of an experimental nature, where operating efficiency is relatively unimportant. And it often happens in the course of systems development, that our initial estimate of depth of structure requirements turns out to be wrong; that is how "kluges" get built. In such instances, the aim at local economies results in a global extravagance.

A partial remedy is for us to become familiar with the spectrum of requirements that systems designers face along this dimension of depth of structure, and to learn (1) how to build adaptable processing modules, and (2) how to tune these to the needs of individual systems. I invite you to join me in the belief that this can and should be done.

Lines of comparable power

Let us consider for a moment the four "corners" of our hypothetical two-dimensional classification space. Since we have no interpretation of negative scope or negative depth of structure, we will locate all systems in the positive quadrant of the two-dimensional plane. At the minimum of both axes, near the origin, we might locate, say, a character-counting program written as a week-end project by a library science student in a mandatory PL/1 course.

High in scope, and low in depth of structure are text editing programs in general. Let us somewhat arbitrarily locate Englebart's editing system in this corner, on the basis of its strong user orientation.

Low in scope, but high in depth of structure: this practically defines Winograd's tabletop robot simulator. The domain of discourse of this system is deliberately severely restricted, but it surpasses any other system I know of in its structural capabilities.

High in both scope and depth of structure: in the real world, no plausible candidate exists. We might imagine this corner of our space filled by a system such as HAL, from the movie "2001"; but nothing even remotely resembling such a system has even been seriously proposed.

The manner in which real-world systems fit into our descriptive space suggests that some kind of trade-off exists between the two dimensions; perhaps it is no accident that the system having the greatest depth-of-structure capabilities is so severely restricted in scope, while the systems having the greatest scope operate at a low level of structural depth. It is my contention that this is indeed not an accident, but that it reflects some important facts about our ability to process textual information automatically. It would seem that, given the current state of the art, we can, as system designers, trade off breadth of scope against advances in structural depth, and vice versa, but that to advance on both fronts at once would require some kind of genuine breakthrough.

This trading relationship between the dimensions can be expressed in terms of *lines of comparable power or* *sophistication.* Having the shape of hyperbolic asymptotes to the axes of our descriptive space, such lines would connect systems whose intrinsic power or sophistication differs only by virtue of a different balance between scope and depth of structure. The state of the art in the field of text processing might then be characterized by the area under the line connecting the most advanced existing systems.

Since genuine breakthroughs are probably not more common in this field than in others, our analysis supports the conclusion that run-of-the-mill system design proposals which promise to significantly extend our automatic text processing capabilities in both scope and depth of structure are probably ill-conceived, or perhaps worse. Yet proposals of this kind are not uncommon, and a number of them attract funds from various sources every year. I feel sure that a better understanding of the dimensions of text processing on the part of sponsoring agencies as well as system designers might result in a healthier and more productive climate of research and development in this field.

Men and machines

Finally, I want to simply mention a set of techniques which can be of inestimable value in breaking through the state-of-the-art barrier in text processing, and to indicate their relation to our two-dimensional descriptive space. I have in mind the set of techniques by which effective man-machine cooperation may be brought to bear in a particular application. It has for some time been known that the human cognitive apparatus possesses a number of powerful pattern-recognition capabilities which have not even been approached by existing computing machinery. A number of projects have investigated the problems of marrying these powers efficiently with the speed and precision of computers to solve problems which neither could manage alone.

In the field of textual data processing, the potential payoff from such hybrid systems, if you will permit me the phrase, increases greatly as we consider higher levels along the dimension of depth of structure. We humans take for granted in ourselves capabilities which astound us in machinery; most 3-year-old children could easily out-perform Winograd's robot simulator, for example. Whereas at the lower levels of this dimension, in tasks like sorting, counting, string replacement, and what not, no man can begin to keep up with even simple machines.

I conclude from these elementary observations that well-designed man-machine systems can greatly extend the scope of systems at the higher end of the dimension of depth of structure, or (to put it in another way) can upgrade the structure-handling capacity of systems having considerable scope. While the design of effective hybrid systems for text processing involves many considerable problems, this approach seems to offer a means of bringing the unique power of computers to bear on applications which now lie on the farther side of the state-of-the-art barrier with respect to fully automatic systems.

ACKNOWLEDGMENTS

The writing of this paper was generously supported by the Center for Computer-Based Behavioral Studies at the University of California at Los Angeles, and was encouraged by the Center's Director, Gerald Shure. I am indebted to Martin Kay and Ronald Kaplan of the RAND Corporation and to J. L. Kuhns of Operating Systems, Inc. for illuminating discussion of the subject matter. The errors are all my own.

REFERENCES

1 *Text editor and corrector reference manual (TECO)*
Interactive Sciences Corporation Braintree Mass 1969

2 N H NIE D H BENT C H HULL
SPSS: Statistical package for the social sciences
McGraw-Hill New York 1970

3 W J DIXON editor
BMD: Biomedical computer programs
University of California Publications in Automatic Computation Number 2 University of California Press Los Angeles 1967

4 D G BOBROW D P MURPHY W TEITELMAN
The BBN-LISP system
Bolt Beranek & Newman BBN Report 1677 Cambridge Massachusetts April 1968

5 *PDP-10 BASIC conversational language manual*
Digital Equipment Corporation DEC-10-KJZE-D Maynard Massachusetts 1971

6 *PDP-10 algebraic interpretive dialogue conversational language manual*
Digital Equipment Corporation DEC-10-AJCO-D Maynard Massachusetts 1970. The AID language in this reference is an adaptation of the RAND Corporation's JOSS language.

7 *QED reference manual*
Com-Share Reference 9004-4 Ann Arbor Michigan 1967

8 *WYLBUR reference manual*
Stanford Computation Center Stanford University Stanford California revised 3rd edition 1970

9 W D ELLIOT W A POTAS A VAN DAM
Computer assisted tracing of text evolution
Proceedings of 1971 Fall Joint Computer Conference Vol 37

10 D C ENGLEBART W K ENGLISH
A research center for augmenting human intellect
Proceedings of 1968 Fall Joint Computer Conference Vol 33

11 O HOLSTI R A BRODY R C NORTH
Theory and measurement of interstate behavior: A research application of automated content analysis
Stanford University May 1964

12 P L WHITE
KWIC/360
IBM Program Number 360D-06.7.(014/022) IBM Corporation St Ann's House Parsonage Green Wilmslow Chesire England United Kingdom

13 M KAY G R MARTINS
The MIND system: The morphological-analysis program
The RAND Corporation RM-6265/2-PR April 1970

14 P J STONE D C DUNPHY M S SMITH D M OGILVIE et al
The general inquirer: A computer approach to content analysis
MIT Press Cambridge 1966

15 E F KELLY
A dictionary-based approach to lexical disambiguation
Unpublished doctoral dissertation Department of Social Sciences Harvard University 1970

16 P STONE M SMITH D DUNPHY E KELLY K CHANG T SPEER
Improved quality of content analysis categories: Computerized disambiguation rules for high frequency English words
In G Gerbner O Holsti K Krippendorf W Paisley P Stone The Analysis of Communication Content: Developments in Scientific Theories and Computer Techniques Wiley New York 1969

17 W A WOODS R M KAPLAN
The lunar sciences natural language information system
Bolt Beranek and Newman Inc Report No 2265 Cambridge Massachusetts September 1971

18 M R QUILLIAN
Semantic memory
In M Minsky editor Semantic Information Processing MIT Press Cambridge Massachusetts 1968

19 B RAPHAEL
SIR: A computer program for semantic information retrieval
In E A Feigenbaum and J Feldman Computers and Thought McGraw-Hill New York 1968

20 C H KELLOGG
A natural language compiler for on-line data management
AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference Vol 33 Part 1 Thompson Book Company Washington D C 1968

21 S C SHAPIRO G H WOODMANSEE
A net-structure based question-answerer: Description and examples
In Proceedings of the International Joint Conference on Artificial Intelligence The MITRE Corporation Bedford Massachusetts 1969

22 S C SHAPIRO
The MIND system: A data structure for semantic information processing
The RAND Corporation R-837-PR August 1971

23 M KAY S SU
The MIND system: The structure of the semantic file
The RAND Corporation RM-6265/3-PR June 1970

24 T WINOGRAD
Procedures as a representation for data in a computer program for understanding natural language
MIT Artificial Intelligence Laboratory MAC TR-84 Massachusetts Institute of Technology Cambridge Massachusetts February 1971

25 R M KAPLAN
The MIND system: A grammar-rule language
The RAND Corporation RM-6265/1-PR March 1970

Social indicators based on communication content

by PHILIP J. STONE

Harvard University
Cambridge, Massachusetts

INTRODUCTION

Early mechanical translation projects served to inject some realism about the complexity of ordinary language processing. While text processing aspirations have become more tempered, today's technology makes possible cost effective applications well beyond the index or concordance. This paper outlines one new challenge that is mostly within curent technology and may be a major future consumer of computer resources.

As we are all aware, industry and government cooperate in maintaining an extensive profile of our economy and its changes. As proposed by Bauer[1] and others, there is a need for indicators regarding the social, in addition to the economic, fabric of our lives. Several volumes, such as a recent one edited by Campbell and Converse,[2] review indexes that can be made on the quality of life. Kenneth Land has documented the growing interest in social indicators, reflected in volumes of reports and congressional testimony, as drawing on a wide basis of support. As the conflicts of the late 1960's and early 1970's within our society made evident the complexity of our own heterogeneous culture, interest in social indicators increased.

Most social indicator discussions focus on statistics similar to economic indicators. A classic case is Durkheim's[4] study on the analysis of suicide rates. Another kind of social indicator, which we consider in this paper, is based on changes in the content of mass media and other public distributions of information, such as speeches, sermons, pamphlets, and textbooks. Indeed, in the same decade as Durkheim's study, Speed[5] compared New York Sunday newspapers between 1891 and 1893, showing how new publication policies (price was reduced from three cents to two cents) were associated with increased attention to gossip and scandal at the expense of attention to literature, religion and politics. Since then, hundreds of such studies, called "content analyses," have been reported.

TYPES OF COMMUNICATION CONTENT INDICATORS

Many different content indicators can be proposed: Which sectors of society have voiced most caution about increasing Federalism? How has the authoritarianism of church sermons changed in different religions? How oriented are the community newspapers to the elites of the community? Such studies, however, can be divided into two major groups.

One group of studies is concerned with comparing the content of different *channels* through which different sectors of society communicate with each other. Such studies often monitor the spread of concepts and attitudes from one node to another in the communication net. Writers such as Deutsch[6] have discussed feedback patterns within such nets.

Another group of studies is based on the realization that a large segment of public media represents different sectors of society communicating with themselves. Social scientists have repeatedly found that people tend to be exposed just to information congruent with their own point of view. Thus, rather than focus on the circulation of information *between* sectors of society, these studies identify different subcultures and look at the content of messages circulated *within* them.

In fact, any one individual belongs to a set of subcultures. On the job, he or she may be exposed to the views of colleagues, while off the job the exposure may be to those with similar leisure time interests, religious preferences, or political leanings. Given the cost effectiveness of television for reaching the mass public, the printed media has become used more for directed messages to different subcultures. Thus, while there has been the demise of general circulation magazines such as the *Saturday Evening Post* and *Look*, the number of magazines concerned with particular trades, hobbies, consumer orientations and levels of literary sophistication has greatly increased.

While the printed media recognizes many different subcultures (and one only has to watch the sprouting of new underground newspapers or trade journals to realize how readily a market can be identified), there has been a more general resistance to recognizing how many subcultures there are and how diverse their views tend to be. Given the enormous complexity of our culture, each sector tends to recognize its own diversity, but assumes homogeneous stereotypes for other sectors. After repeated blunders, both the press and the public are coming to realize that there are many different subcultures within the black community, the student community, the agricultural community, just as we all know there are many different subcultures in the computer community. As sociologist Karl Mannheim[7] identified some years ago, the need to monitor our culture greatly increases with such heterogeneity.

Gradually, awareness of a need is turning into action. Since the *Behavioral and Social Science* (Bass) report[8] released by the National Academy of Science in 1969 gave top priority to developing social indicators, government administration has been set up to coordinate social indicator developments and several large grants have been issued. Within coming years, we may expect significant sums appropriated for social indicators.

COMPARISON WITH CONTENT ANALYSIS RESEARCH

What language processing expertise do we have today to help produce such social indicators? The Annual Review prepared by the American Documentation Institute or reports on such large text processing systems as Salton's "SMART"[9] offer a wide variety of possibly relevant procedures. The discussion here focuses on techniques developed explicitly for content analysis.

Content analysis procedures map a corpus of text into an analytic framework supplied by the investigator. It is information reducing, in contrast to an expansion procedure like a concordance, in that it discards those aspects of the text not relevant to the analytic framework. As a social science research technique, content analysis is used to count occurrences of specific symbols or themes.

The main difference between content analysis as a social science research technique and mass media social indicators concerns sampling. A researcher samples text relevant to hypotheses being tested. Only as much text need be processed as necessary to substantiate or disconfirm the hypothesis. Usually, this involves thousands or tens of thousands of words of text. A social indicators project, on the other hand, involves monitoring many different text sources over what may be long periods of time. The total text may run into millions of words.

A hypothetical example illustrates how a social indicators project can come to be such a large size. A social indicators project may compare a number of different subcultural sectors in each of several different geographic locations. Within each sector, the sample should include several media, so it does not reflect the biases of one originator. The monitoring might cover several decades, with a new indicator made bimonthly. Imagine then a 4 dimensional matrix representing 14 (subcultural sectors) × 5 (geographic regions) × 4 (originating sources) × 150 (bimonthly periods). Each cell of this matrix might contain a sample of 15,000 words of text. The result is a text file of over a billion characters.

Social science content analysis, which has been computer aided for over a decade (see for example, Stone et al.,[10] Stone et al.,[11] Gerbner et al.[12]), has used manual keypunching to provide the modest volumes of machine readable text needed. If the content analysis task were simple (such as in our first example below), human coders were often less expensive than the cost of getting the text to machine readable form. Computer aided content analysis has tended to focus on those texts, such as anthropologists' files of folktales, where the same material may be intensively studied to test a variety of hypotheses.

Social indicators of public communications, on the other hand, will require high speed optical readers capable of handling text in a wide variety of printing fonts. Optical readers for selected fonts have been around for some time, but readers capable of adapting to many new fonts are just coming into existence. The large general purpose reader developed by Information International, which incorporates a PDP-10 as part of its hardware, represents this kind of machine. It is able to "learn" new fonts and then offer high speed reading from microfilm with a low error rate, even of third generation Xerox copy.

Both social science content analysis research and social indicators allow for some noise, just as economic indicators tolerate an error factor. Some of the noise stems from sampling procedures. Other noise comes from measurement procedures. The "quality control" of social science research or monitoring procedures involves keeping the noise to a tolerable minimum. Assurances are also needed that the noise is indeed random, rather than leading to specifiable biases. With large sampling procedures, a tolerable modest random noise level should be considerably more than allowed in many kinds of text processing applications. For example, a single omission in an automated document classification scheme might cause a very important document to go unnoticed by the users of the information retrieval system, thus causing great loss.

LEVELS OF COMPLEXITY

Both content analysis procedures for testing hypotheses and procedures for creating social indicators come at varying levels of complexity. Some pose little difficulty for today's text processing capabilities while others pose major challenges. If one accepts a growing consensus among artificial intelligence experts that a successful language translation machine must "understand," in a significant sense of that word, the subject matter it is translating, then the most complicated social indicator tasks begin to approach this domain of challenge. Let us start at the simpler levels, showing how these needs have been met in content analysis, and work up to these more difficult challenges.

The simplest measure is to identify mentions of an individual, place, group, or race. For example, Johnson, Sears and McConahay[13] performed a manual content analysis of what they call "black invisibility" since the turn of the century in the Los Angeles press. They show that the percent of newspaper space devoted to blacks in the major papers is much less than their percent of the Los Angeles population warrants, and, furthermore, the ratio has been getting worse over time. A black person can read the Los Angeles press and obtain the impression that blacks do not exist there. Thus, point out the authors, some blacks took a "We won!" attitude toward the large amount of destruction in the Watts riots. Why? As reported by Martin Luther King,[14] they said "We won because we made them pay attention to us." There is indeed a hunger to have one's existence recognized.

"Black invisibility" can be assessed by counting the number of references to blacks compared to whites, or the newspaper column inches given to each. A computer content analysis would need a dictionary of names referring to black persons or groups. The computer should have an ability to automatically update this dictionary as processing continued, for race may be only identified with early newspaper stories about the person or group. Thus, few stories today about Angela Davis any more identify her race. The computer challenge, should optical readers have had the text ready for processing, would have been minimal.

Johnson, Sears, and McConohay carried their research another step, classifying the stories according to whether they dealt with anti-social activities, black entertainers, civil rights, racial violence and several other categories. These additional measures make "black invisibility" more evident, for what little coverage blacks receive is often unfavorable and unrepresentative of the community. These additional topic identifications would again hold little difficulty for a computer analysis. It is not difficult to recognize a baseball story, a violent crime story, or a society event.

The Johnson, Sears and McConahay "black invisibility" indexes were only made on two newspapers with very limited samples in the earlier part of the time period studied. Their techniques, however, could be applied to obtain "black invisibility" indexes for both elite and tabloid press in every major metropolitan area of the country. It is an example of how a content analysis measure can have considerable potential as a future social indicator.

The next level of complexity is represented by what we call thematic analysis. For example, we might be interested in social indicators measuring attitudes toward increasing Federalism in our society. Separate indicators might be developed to tap occurrences of themes such as the following:

(1) The Federal government as an appropriate recipient of complaints about . . .
(2) The Federal government as initiator of programs for . . .
(3) The Federal government as restricting or controlling . . .
(4) The Federal government as responsible for the well being of . . .

Such themes are measured by first identifying synonyms. Rather than refer to the "Federal government," the text may refer to a particular agency. The verb section may have alternative forms of expression. F inally separate counts might be kept for each theme relevant to different target clusters such as agriculture, industry, minority groups, consumer goods, etc.

Past work in content analysis has offered considerable success in studies on a thematic level. Thus Ogilvie (1966) found considerable accuracy in computer scoring of "need achievement" themes in stories made up by subjects. The scoring involved thematic identifications similar in complexity to the Federalism measures cited above. Ogilvie found that the correlation between the computer and a human coder was about .85, or as high as the correlation between two human coders.

A still higher complexity is represented by the packaging of thematic statements into an argument, plot, or rationale. This has recently become prominent in psychology, with Abelson[16] writing about "opinion molecules" while Kelly[17] writes of "causal schemata."

The concern is with, if I may substitute computer terminology, various "subroutines" that we draw on to explain the world about us. Many such subroutines are shared by the community at large, so that a passing reference to any part of the subroutine can be expected to cause the listener to invoke the whole subroutine. To take a very simple example, such phrases as a "Communist inspired plot," "subversive action," and "Marxist goals" can all be taken as invoking a highly shared molecule including something as follows:

> Communists create (inspire) plots involving subversive actions against established ways in order to force changes in society toward their Marxist goals.

Matters are rather simple when dealing with such weatherbeaten old molecules, but take the end run kinds of debates between politicians about school bussing to try and identify the variety of molecules surrounding that topic. The inference of underlying molecules involves theoretical issues that can go well beyond text processing problems.

Again, there is a relevant history in content analysis, although computer aided procedures have only recently had any successes. The classic manual study was by Propp[18] who showed that Russian folktales fell into variants of a basic plot. Recently, anthropologists such as Colby[19] and Miranda[20] have pushed further the use of the computer to study folktale plots. Investigators such as Shneidman[21] have worked on detailed manual techniques to identify the forms of "psycho-logic" we use in everyday explanations. Social indicators at this level should pose considerable difficulty for some time to come.

NEW TEXT PROCESSING RESOURCES

Content analysis research may share with social indicators projects in the priorities for new text processing resources. These priorities may be quite different from those in information retrieval or other aspects of text processing. We here review these priorities as we see them.

While automated linguistic analysis has been preoccupied with questions of syntactic analysis, content analysis work has given priority to semantic accuracy. Semantic errors effect even the simplest levels of measurement and were known to cause considerable noise in many measurement procedures. Even the "black invisibility" study, for example, is going to have to be able to distinguish between "black" the color and "black" the race, as well as other usages of "black." A Federalism study may expect verbs like "restrict" and "control," but in fact the text may use such frightfully ambiguous words as "run," "handle" or "order." A first order of business has been to reduce such noise to more manageable levels.

One might argue that procedures for such semantic identifications should come after the text has received a syntactic analysis. Certainly this would simplify the task and increase accuracy. However, many simpler social indicators and content analysis tasks do not otherwise need syntactic analyses. For social indicator projects, the large volumes of text discourage invoking syntactic routines unless they are really needed. In content analysis research, text is often transcripts of conversational material having a highly degenerate syntactical form. For these applications, a syntactically dependent analysis of word senses might be less than satisfactory. Thus, for both social indicators and content analysis research, it makes sense to attempt identification of word senses apart from syntactic analysis.

A project was undertaken some five years ago to develop computer routines that would be able to identify major word senses for all words having a frequency of over 40 per million. This criterion resulted in a list of 1815 entries covering about 90 percent of running text. Identification of real, separate word senses is a thorny problem we have discussed elsewhere; let it simply be pointed out here that the number of word senses in a dictionary tends to be directly proportional to the size of the dictionary. Our goal was to cover the basic distinctions (such as "black" the race vs "black" the color) rather than many fine-graded distinctions (such as those of a word like "fine").

Of the 1815 candidates, some 1200 were identified as having multiple meanings. Two thirds of these, or about 800 words offered considerable challenge for word sense identifications. Rules for identifying word senses were developed for each of these multiple meaning words. Each rule could test the word environment (specifying its own boundary parameters) for the presence or absence of particular words or any of sixty different markers. Each rule, written in a form suitable for compilation by a weak precedence grammar, could either assign senses or transfer to other rules, depending on the outcome. The series of rules used for testing any one word thus formed a routine.

The implementation of these rules on a computer emphasized efficiency. Since marker assignments often depended on word senses being identified, deadlocks could occur with some rules testing for markers on neighboring words which could not yet be assigned until the word in question was resolved. Strategies include the computer looking into dictionary entries to see if the marker category is among the possible outcomes. Despite such complicated options, occasionally resulting in multiple passes, the word sense procedures are remarkably fast, to the point of being feasible for social indicators work.

The accuracy of the word sense identification procedures was tested on a 185,000 word sample drawn both from Kucera and Frances[22] and our own text files. A variety of tests were performed. For example, for a sample of 671 particularly difficult homographs, covering 64,253 tokens in the text, correct assignment was made 59,716 times or slightly over 92 percent of the time. The procedures thus greatly reduce the noise in word sense assignments.

The second priority for even some of the simplest social indicator projects should be pronoun identification. The importance of the problem depends on the kinds of tabulations that are to be made. If the question is whether any mention is made in the article, then pronouns are not such a crucial issue. If the question involves counting how many references were made, then references should be identified in both noun and pronoun form.

We believe that more work should be encouraged on pronoun identification so as to be better prepared for future social indicators research. Because many pronouns involve references outside the sentence, the problem is beyond most current syntax studies. Winograd[23] provides a heartening example of how well pronoun identification can be made for local discourse on a specific topic area.

The third priority is syntactic analysis for thematic identification purposes. This is not just a general syntactic analysis, but an analysis to determine if the text matches one of the thematic templates relative to a social indicator. Large amounts of computer time can be saved by only calling on the syntactic routine after it is established that all the semantic components relevant to the theme are indeed present. Syntactic analysis can stop as soon as it is established that the particular word order cannot be an example of that theme. In general, we find that a case grammar is most akin to thematic analysis needs.

The transition network approach of Woods[24] holds considerable promise for such syntactic capabilities. Gary Martins, who is with us on the panel, is already exploring the application of such transition networks to content analysis problems. This work should be of considerable utility in the development of social indicators based on themes.

Finally, we come to the need for inference systems to handle opinion molecules and the like. Such devices as Hewitt's PLANNER[25] may have considerable utility for such social indicator projects. A PLANNER operation includes a data base and a set of theorems. Given a text statement, PLANNER can attempt to tie it back through the theorems until a match is made in the data base. For any editorial, for example, the successful theorem paths engendered could identify which molecules were being invoked and their domain of application. At present, this is but conjecture; much work needs to be done.

These priorities, then, are explicitly guided by what Weizenbaum[26] calls a "performance mode," in this case toward creating useful social indicators. They may well conflict with text processing priorities in computational linguistics or artificial intelligence. Some social indicators may only be produced in the distant future, but meanwhile important results can be accomplished using optical readers and current text processing sophistication.

DEVELOPING SOCIAL INDICATOR TEXT ARCHIVES

Having considered text processing research priorities, let us examine what concurrent steps are needed if relevant text files are to be created and put to effective use.

At present, our archiving of text material is haphazard and, for social indicator purposes, subject to major omissions. The American public (as publics in most advanced societies) spends more than four times as many hours watching television compared to all forms of reading put together (Szalai, et al.[27]). Yet, even with the incredible salience of network evening newscasts or documentary specials, the networks are not required to place television scripts in a public archive. A content analysis like Efron's *The News Twisters*[28] had to be made from homemade tape recordings of news broadcasts.

Similarly if one is to study the content of cummunication channels between sectors of society, one needs both original and intermediate sources such as press releases and wire service transmissions. Past critics of our news media such as Cirino[29] have had to make extensive efforts to obtain the necessary primary information. Better central archiving is very much needed.

As discussed by Firestone,[30] considerable attention will have to be given to coordinating the sampling of text with sampling used for other social indicators. For example, it makes obvious sense to target the sampling of union newsletters to correspond to union memberships selected for repeated survey interviews. In one of our own studies, Stone and Brody[31] compared a content analysis of news stories on the Vietnam war with the results of Gallup survey questions about the effectiveness of the president. This study would have been greatly aided by (a) better text files of representative news stories from across the nation and (b) survey information as to media exposure. With less adequate data, the quality of the analysis suffers.

SAFEGUARDING THE PUBLIC

On the one hand, since the files are based on public communications, investigators outside the government should have access to the archives for testing different models. In this sense, such files would be similar to the computer economic data bases for testing econometric models now made available by commercial organizations.

On the other hand, the same technology used to produce social indicators based on content can be used to invade the content of private communications. This author, for one, is worried about current military sponsored research that aims to make possible a computer monitoring of voice grade telephone communication. After all that has been written about privacy, a much closer safeguard is needed. Further work on content analysis techniques must be coordinated with such safeguards.

SUMMARY

This paper has outlined how computer text processing resources may be used to produce social indicators of communication content. A new challenge of considerable scale is forecast. The relations of such indicators to existing content analysis research techniques is identified. Priorities based on social indicator requirements are offered for future text processing research. Because of the scale of the operation and its distinct requirements, we suggest that social indicators based on communication content be considered separate from other computer text processing applications. Immediate attention is needed for text archiving and safeguarding the privacy of communications.

REFERENCES

1 R BAUER
Social indicators
MIT Press 1966

2 A CAMPBELL P CONVERSE (ed)
The human meaning of social change
Russell Sage Foundation 1972

3 K LAND
Social indicators
In R Smith Social Science Methods—A New Introduction Vol 2 In Press

4 E DURKHEIM
Suicide—A study in sociology
1897 (Trans from the French 1951 Free Press)

5 J G SPEED
Do newspapers now give the news?
The Forum 1893 Vol 15 pp 705-711

6 K DEUTSCH
Nerves of government
Free Press 1963

7 K MANNHEIM
Ideology and utopia—An introduction to the sociology of knowledge
1931 (English translation: Harcourt)

8 NATIONAL ACADEMY OF SCIENCE
Behavioral and social sciences—Outlook and needs
Prentice Hall 1969

9 G SALTON
The SMART retrieval system
Prentice Hall 1971

10 P STONE R BALES J Z NAMENWIRTH D OGILVIE
The general inquirer
Behavioral Science 1962 Vol 7 pp 484-498

11 _____ D C DUNPHY M S SMITH D M OGILVIE
The general inquirer—A computer approach to content analysis
MIT Press 1966

12 G GERBNER O HOLSTI K KRIPPENDORFF W PAISLEY P J STONE
The analysis of communications content
Wiley Press 1969

13 P B JOHNSON D SEARS J McCONAHAY
Black invisibility, the press and the Los Angeles riot
Amer J Sociology 1971 Vol 76 pp 698-721

14 M L KING
Where do we go from here? Chaos or community?
Beacon Press 1967

15 D M OGILVIE
In P Stone et al op cit pp 191-206

16 R P ABLESON
Psychological implication
In R P Abelson E Aronson W McGuire T Newcomb M Rosenberg and P Tannenbaum Theories of Cognitive Consistency Rand McNally 1968

17 H KELLY
Causal schemata and the attribution process
General Learning Press 1972

18 V PROPP
Morphology of the folktale
1927 American Folklore Society (Trans 1958)

19 B N COLBY
Folk narrative
Current Trends in Linguistics Vol 12 1972

20 P MIRANDA
Structural strength, semantic depth, and validation procedures in the analysis of myth
Proceedings Quatrieme Symposium sur les Structures Narratives Konstanz Germany 1971 In Press

21 E S SHNEIDMAN
Logical content analysis: An explication of styles of "concludifying"

In Gerbneret al op cit

22 H KUCERA W FRANCES
Computational analysis of present-day American English
Brown University Press 1967

23 T WINOGRAD
Procedures as a representation for data in a computer program for understanding natural language
Report MAC TR-84 MIT February 1971 (Selections reprinted in Cognitive Psychology 1972 #1)

24 W WOODS
Transitional network grammars for natural language analysis
Comm ACM 1970 Vol 13 pp 591-602

25 C HEWITT
PLANNER—A language for proving theorems in robots
Proc of IJCAI 1969 pp 295-301

26 J WEIZENBAUM
On the impact of the computer on society
Science Vol 176 pp 609-614 1972

27 A SZALAI E SCHEUCH P CONVERSE P STONE
The use of time
Mouton 1972

28 E EFRON
The news twisters
Nash 1971

29 R CIRINO
Don't blame the people
Diversity Press 1971

30 J M FIRESTONE
The development of social indicators from content analysis of social documents
Policy Sciences In Press

31 P STONE R BRODY
Modeling opinion responsiveness to day news—The public and Lyndon Johnson 1965-1968
Social Science Information Vol 9 #1 pp 95-122

The DOD COBOL compiler validation system

by GEORGE N. BAIRD

Department of the Navy
Washington, D. C.

INTRODUCTION

The ability to benchmark or validate software to ensure that design specifications are satisfied is an extremely difficult task. Test data, generally designed by the creators of said software, is generally biased toward a specific goal and tend not to cover many of the possibilities of combinations and interactions. The philosophy of suggesting that "a programmer will never do . . ." or "this particular situation will never happen" is altogether absurd. First, "never" is an extremely long time and secondly, the Hagel theorem of programming states that "if it can be done, whether absurd or not, one or more programmers will more than likely try it."

Therefore, if a particular piece of software has been thoroughly checked against all known extremes and a majority of all syntactical forms, then the Hagel theorem of programming will not affect the software in question. The DOD CCVS attempts to do just that by checking for the fringes of the specifications of X3.23-1968[1] and known limits. It is assumed that a COBOL compiler will perform satisfactorily for the audit routines, then it is likely that the compiler supports the entire language. However, if the computer has trouble with handling the routines in the CCVS it can be assumed that there will indeed be other errors of a more serious nature.

The following is a brief account of the history of the DOD CCVS, the automation of the system and the adaptability of the system to given compilers.

BACKGROUND

The first revision to the initial specification for COBOL (designated as COBOL-1961[2]) was approved by the Executive Committee of the Conference on Data Systems Languages* and published in May of 1961. Recognizing that the language would be subject to additional development and change, an attempt was made to create uniformity and predictability in the various implementations of COBOL compilers. The language elements were placed in one of two categories: required and elective.

Required COBOL-1961 consisted of language elements (features and options) which must be implemented by any implementor claiming a COBOL-1961 compiler. This established a common minimum subset of language elements for COBOL compilers and hopefully a high degree of transferability of source programs between compilers if this subset was adhered to.

Elective COBOL-1961 consisted of language elements whose implementation had been designated as optional. It was suggested that if an implementor chose to include any of these features (either totally or partially) he would be expected to implement these in accordance with the specifications available in COBOL-1961. This was to provide a logical growth for the language and attempt to prevent a language element from having contradictory meaning between the language development specifications and implementor's definition.

As implementors began providing COBOL compilers based on the 1961 specifications, unexpected problems became somewhat obvious. The first problem was that the specifications themselves suggested mandatory as well as optional language elements for implementing COBOL compilers. In addition the development docu-

* The Conference on Data Systems Languages (CODASYL) is an informal and voluntary organization of interested individuals supported by their institutions who contribute their efforts and expenses toward the ends of designing and developing techniques and languages to assist in data systems analysis, design, and implementation. CODASYL is responsible for the development and maintenance of COBOL.

ment produced by CODASYL was likely to change periodically thus, providing multiple specifications to implement from. Compilers could consist of what the implementor chose to implement which would severely handicap any chance of transferability of programs among the different compilers, particularly since no two implementors necessarily think alike. Philosophies vary both in the selection of elements for a COBOL compiler and in the techniques of implementing the compiler itself. (As ridiculous as it may sound, some compilers actually scan, syntax check and issue diagnostics for COBOL words that might appear in comments both in the REMARKS paragraph of the Identification Division and in NOTE sentences in the Procedure Division.) The need for a common base from which to implement became obvious. If the language was to provide a high degree of compatability, then all implementations had to be based on the same specification.

The second problem was the reliability of the compiler itself. If the manual for the compiler indicated that it supported the DIVIDE statement, the user assumed this was true. If the compiler then accepted the syntax of the DIVIDE statement, the user assumed that the object code necessary to perform the operation was generated. When the program executed, he expected the results to reflect the action represented in his source code. It appears that in some cases perhaps no code was generated for the DIVIDE statement and the object program executed perfectly except for the fact that no division took place. In another case, when the object program encountered the DIVIDE operation, it simply went into a loop or aborted. At this point, the programmer could become decidedly frustrated. The source code in his program indicated that: (1) he requested that a divide take place, (2) there was no error loop in his program, (3) the program should not abort. This is the problem we are addressing: A programmer should concern himself with producing a source program that is correct logically and the necessary operating system control statements to invoke the COBOL compiler. In doing so, he should be able to depend on the compiler being capable of contributing its talent in producing a correct object program.

If the user was assured that either: (1) each instruction in the COBOL language had been implemented correctly, or, (2) that each statement which was implemented did not give extraneous results, then the above situation could not exist.

Thus, the need for a validation tool becomes apparent. Although all vendors exercise some form of quality control on their software before it is released, it is clear that some problems may not be detected. (The initial release of the Navy COBOL audit routines revealed over 50 bugs in one particular compiler which had been released five years earlier.)

By providing the common base from which to implement and a mechanism for determining the accuracy and correctness of a compiler relative to the specification, the problem of smorgasbord compilers (that may or may not produce expected results) should become extinct.

The standardization of COBOL began on 15 January 1963. This was the first meeting of the American Standards Association Committee, X3.4.4,* the Task Group for Processor Documentation and COBOL. The program of work for X3.4.4 included . . . "Write test problems to test specific features and combinations of features of COBOL. Checkout and run the test problems on various COBOL compilers." A working group (X3.4.4.2) was established for creating the "test problems" to be used for determining feature availability.

The concept of a mechanism for measuring the compliance of a COBOL compiler to the proposed standard seemed reasonable in view of the fact that other national standards did indeed lend themselves to some form of verifications, i.e., 2×4's, typewriter keyboards, screw threads.

IMPLEMENTING A VALIDATION SYSTEM FOR COBOL

In order to implement a COBOL program on a given system, regardless of whether the program is a validation routine or an application program, the following must be accomplished:

1. The special characters used in COBOL (i.e., '(', ')', '*', '+', '<' etc.) must be converted for the system being utilized.†
2. All references to implementor-names within each of the source programs must be resolved.
3. Operating System Control Cards must be pro-

* The American Standards Association (ASA), a voluntary national standards body evolved to the United States of America Standards Institute (USASI) and finally the American National Standards Institute (ANSI). The committee X3.4.4 eventually became X3J4 under a reorganization of the X3 structure. X3J4 is currently in the process of producing a revision to X3.23-1968.

† For most computers the representatives for the characters A-Z, 0-9, and the space (blank character) are the same. However, there is sometimes a difference in representation of the other characters and therefore conversion of these characters from one computer to another may be necessary.

duced which will cause each of the source programs to be compiled and executed. Additionally, the user must have the ability to make changes to the source programs, i.e., delete statements, replace statements, and add statements.

4. As the programs are compiled, any statements that are not syntactically acceptable to the compiler must be modified or "deleted" so that a clean compilation takes place and an executable object program is produced.

5. The programs are then executed. All execution time aborts must be resolved by determining what caused the abort and after deleting or modifying that particular test or COBOL element, repeating steps 3 and 4 until a normal end of job situation exists.

Development of audit routines

March 1963, X3.4.4.2 (the Compiler Feature Availability Working Group) began its effort to create the COBOL programs which would be used to determine the degree of conformance of a compiler to the proposed standard. The intent of the committee was not to furnish a means for debugging compilers, but rather to determine "feature availability." Feature availability was understood to mean that the compiler accepted the syntax and produced object code to produce the desired result. All combinations of features were not to be tested; only a carefully selected sample of features (singly and in combination) were to be tested to insure that they were operational. The test programs themselves were to produce a printed report that would reflect the test number and when possible whether the test "Passed" or "Failed." See Figure 1.

When a failure was detected on the report, the user could trace the failure to the source code and attempt to identify the problem. The supporting code (printing routine, pass routine, fail routine, etc.) was to be written using the most elementary statements in the low-level of COBOL. The reason for this was twofold:

1. The programs would be able to perform on a minimum COBOL compiler (Nucleus level 1, Table Handling level 1, and Sequential Access level 1).
2. The chances of the supporting code not being acceptable to the compiler being tested were lessened.

```
Source Statements

TEST-0001.

   MOVE 001 TO TEST-NO.
   MOVE ZERO TO ALPHA.
   ADD 1 TO ALPHA.
   IF ALPHA = 1 PERFORM PASS ELSE PERFORM FAIL.

TEST-0002.
Results

   TEST    NO    P  -  F
   ADD      1    P
     .
     .
     .
   ADD     21          F
```

Figure 1—Example of X3.4.4.2 test and printed results

The programs, when ready, would be provided in card deck form along with the necessary documentation for running them. (The basic philosophies of design set forth by X3.4.4.2 were carried through all subsequent attempts to create compiler validation systems for COBOL.)

Assignments were made to the members of the committee and the work began. This type of effort at the committee level, however, was not as productive as the work of standardizing the language itself.

In April 1967, the Air Force issued a contract for a system to be designed and implemented which could be used in measuring a compiler against the standard. The Air Force COBOL Compiler Validation System was to create test programs and adapt them to a given system automatically by means of fifty-two parameter cards.

The Navy COBOL audit routines

In August of 1967, The Special Assistant to the Secretary of the Navy created a task group to influence the use of COBOL throughout the Navy. Being aware of both the X3.4.4.2 and Air Force efforts, (as well as the time involved for completion), a short term project was established to determine the feasibility of validating COBOL compilers. After examining the information and test programs available at that time, the first set of routines was produced. In addition to the original X3.4.4.2 philosophy, the Navy added the capability of providing the result created by the computer as well as the expected result when a test failed. Also, instead of a test number, the actual procedure name in the source program was reflected in the output. See Figure 2.

The preliminary version of the Navy COBOL audit routines was made up of 12 programs consisting of about 5000 lines of source code. The tailoring of the programs to a particular compiler was done by hand (by physically changing cards in the deck or by using the vendor's software for updating COBOL programs). As tests were deleted or modified, it was difficult to bring the programs back to their virgin state for subsequent runs against different compilers or for determining what changes had to be made in order that the programs would execute.

This was a crude effort, but it established the necessary evidence that the project was feasible to continue and defined techniques for developing auditing systems. Because of the favorable comments received on this initial work done by the Navy, it appeared in the best interest of all to continue the effort.

After steady development and testing for a year, Version 4 of the Navy COBOL Audit Routines was released in December 1969. The routines consisted of 55 Programs, consisting of 18,000 card images capable of testing the full standard. The routines had also become one of the benchmarks for all systems procured by the Department of the Navy in order to ensure that the compiler delivered with the system supported the required level of American National Standard COBOL.*

Also, Version 4 introduced the VP-Routine, a program that automated the audit routines. Based on fifty parameter cards, all implementor-names could be resolved and the test programs generated in a one-pass operation. See Figure 3.

In addition, by coding specific control cards in the Working-Storage Section of the VP-Routine as constants, the output of the VP-Routine became a file that very much resembled the input from a card reader, i.e., control cards, programs, etc.

By specifying the required Department of Defense COBOL subset of the audit routines to be used in a validation, only the programs necessary for validating that subset of elements or modules would be selected, i.e., SUBSET-A, B, C, or D. The capability also existed to update the programs as the "card reader" file was being created. The use of the VP-Routine was not mandatory at this time, but merely to assist the person validating the compiler in setting up the programs for compilation. Once the VP-Routine was set up for a given system, there was little trouble running the audit routines. The user then had only to concern himself with the validation itself and with achieving successful results from execution of the audit routines. When an updated set of routines was distributed, there was no effort involved in replacing the old input tape to the VP-Routine with the new tape.

Source Statements

```
  ADD-TEST-1.
    MOVE 1 TO ALPHA.
    ADD 1 TO ALPHA.
    IF ALPHA =2 PERFORM PASS ELSE PERFORM FAIL.
      .
      .
      .
```

Results

```
  FEATURE  PARAGRAPH   P/F   COMPUTED  EXPECTED
  ADD      ADD-TEST-1  FAIL  1         2
  ADD      ADD-TEST-2  PASS
```

Figure 2—Example of Navy test and printed results

V-P Routine Input:

```
X-0  SOURCE-COMPUTER-NAME
X-1  OBJECT-COMPUTER-NAME
X-3
 .
 .
 .
X-8  PRINTER
X-9  CARD-READER
X-10
 .
 .
 .
 .
X-50
```

Audit Routine File:

```
SOURCE-COMPUTER.
     XXXXX0
         .
         .
         .
     SELECT PRINT-FILE ASSIGN TO
     XXXXX8
```

The audit routine after processing would be:

```
SOURCE-COMPUTER.
     SOURCE-COMPUTER-NAME.
         .
         .
         .
     SELECT PRINT-FILE ASSIGN TO
     PRINTER.
```

Figure 3—Example of input to the support routine, Population file where audit routines are stored and resolved audit routine after processing

The Air Force COBOL audit routines

The Air Force COBOL Compiler Validation System (AFCCVS) was not a series of COBOL programs but rather a test program generator. The user could select

* In 1968, the Department of Defense, realizing that several thousand combinations of modules/levels were possible, established four subsets of American National Standard COBOL for procurement purposes.

```
Source statement in test library
T 1N078A101NUC, 2NUC                                          4U
400151  77  WRK-DS-18V00                PICTURE  S9(18).
400461  77  A180NES-DS-18VOO            PICTURE  S9(18).
400471                                VALUE 11111111111111111.
400881  77  A18ONES-CS-18VOO            PICTURE S9(18) COMPUTATIONAL
400891                                VALUE 111111111111111111.
802925  TEST-1NUC-078.
802930      MOVE A18ONES-DS-18VOO                       TO WRK-DS-18VOO.
802935      ADD A180NES-CS-18VOO                        TO WRK-DS-18VOO
802940      MOVE WRK-DS-18VOO                           TO SUP-WK-A.
802945      MOVE '222222222222222222'                   TO SUP-WK-C.
802950      MOVE '1NO78'                                TO SUP-ID-WK-A
802955      PERFORM SUPPORT-RTN THRU SUP-TRN-C.
Test results
  .1NO78            .1NO79.
  .222222222222222222.09900.
  .                 .____.
```

Figure 4—Example of Air Force test and printed results

the specific tests or modules he was interested in and the AFCCVS would create one or more programs from a file of specific tests which were then compiled as audit routines. Implementor-names were resolved as the programs were generated based on parameter cards stored on the test file or provided by the user.

The process required several passes, including the sorting of all of the selected tests to force the Data Division entries into the Data Division and place the tests themselves in the Procedure Division where they logically belonged. An additional pass was required to eliminate duplicate Data Division entries (more than one test might use the same data-item and therefore there would be more than one copy in the Data Division). See Figure 4.

Still another program was used to make changes to the source programs as the compiler was validated. As in the Navy system, certain elements had to be eliminated because: (1) they were not syntactically acceptable to the compiler or, (2) they caused run time aborts.

Department of Defense COBOL validation system

In December 1970, The Deputy Comptroller of ADP in the Office of the Secretary of Defense asked the Navy to create what is now the DOD Compiler Validation System for COBOL taking advantage of: (1) the better features of both the Navy COBOL Audit Routines (Version 4) and the Air Force CCVS and (2) the four years of in-house experience in designing and implementing audit routines on various systems as well as the actual validation of compilers for procurement purposes.

The Compiler Validation System (of which the support program was written in COBOL) had to be readily adaptable to any computer system which supported a COBOL compiler and which was likely to be bid on any RFP issued by the Department of Defense or any of its agencies. It also had to be able to communicate with the operating system of the computer in order to provide an automated approach to validating the COBOL compiler. The problem of interfacing with an operating system may or may not be readily apparent depending on whether an individual is more familiar with IBM's Full Operation System (OS), which is probably the most complex operating system insofar as establishing communication between itself and the user is concerned, or with the Burroughs Master Control Program (MCP), where the control language can be learned in a fifteen or twenty minute discussion.

Since validating a compiler may not be necessary very often, the amount of expertise necessary for communicating with the CVS should be kept to a minimum. The output of the routines should be as clear as possible in order not to confuse the reviewer of the results or to suggest ambiguities.

The decision was made to adopt the Navy support system and presentation format for several reasons. (1) It would be easier to introduce the Air Force tests into the Navy routines as additional tests because the Navy routines were already in COBOL program format. It would have been difficult to recode each of the Navy tests into the format of specific tests on the Air Force Population File because of the greater volume of tests. (2) The Navy support program had become rather versatile in handling control cards, even for IBM's OS, whereas the Air Force system had only limited control card generation capability.

The merging of the Air Force and Navy routines

The actual merging of the routines started in February 1971 and continued until September 1971. During the merging operation, it was noted that there was very little overlap or redundancy in the functions tested by the Air Force and Navy systems. In actuality, the two sets of tests complemented each other. This could only be attributed to the different philosophies of the two organizations which originally created the routines. For example in the tests for the ADD statement:

Air Force	*Navy*
signed fields	unsighed fields
most fields 18 digits long	most fields 1-10 digits long
more computational items	more display items

After examining the Add tests for the combined DOD routines, it was noticed that a few areas had been totally overlooked.

1. An ADD statement that forced the "temp" used by the compiler to hold a number greater than 18 digits in length:

   ```
   i.e., ADD    +999999999999999999
                +999999999999999999
                +999999999999999999
                −999999999999999999
                −999999999999999999
                −99   TO ALPHA
   ```

   . . . where the intermediate result would be greater than 18 digits, but the final result would be able to fit in the receiving field.
2. There were not more than eight operands in any one ADD test.
3. A size error test using a COMPUTATIONAL field when the actual value could be greater than the described size of the field, i.e., ALPHA PICTURE 9(4) COMP. . . specifies a data item that could contain a maximum value of 9999 without an overflow condition; however, because the field may be set up internally in binary, the decimal value may be less than the maximum binary value it could hold:

   Maximum COBOL value = 9999
   Maximum hardware value≃16383

Therefore, from this point of view, the merging of the routines disclosed the holes in the validation systems being used prior to the current DOD routines.

```
Source statements

ADD-TEST-1.
   MOVE 1 TO ALPHA.                      Initialization if
                                           necessary.
   ADD 1 TO ALPHA.                       The Test.
   IF ALPHA = 2                          Check the results of the
       PERFORM PASS                        test and handle the
                                           accounting of that
                                           test.
    ELSE
       GO TO ADD-FAIL-1.
   GO TO ADD-WRITE-1.                    Normal exit path to the
                                           write paragraph.
ADD-DELETE-1.                            Abnormal path to the
   PERFORM DELETE.                         write statement if the
   GO TO ADD-WRITE-1.                      test is deleted via the
                                           NOTE statement.
ADD-FAIL-1.                              Correct and computed
   MOVE ALPHA TO COMPUTED.                 results are formatted
   MOVE '2' TO CORRECT.                    for printing.
   PERFORM FAIL.
ADD-WRITE-1.                             Results are printed.
   MOVE 'ADD-TEST-1' TO PARAGRAPH-NAME.
   PERFORM PINT-RESULTS.
ADD-TEST-2.
```

Figure 5—Example of DOD test and supporting code

The general format of each test is made up of several paragraphs: (1) the actual "test" paragraph; (2) a "delete" paragraph which takes advantage of the COBOL NOTE for deleting tests which the compiler being validated cannot handle; (3) the "fail" paragraph for putting out the computed and correct results when a test fails; and (4) a "write" paragraph which places the test name in the output line and causes it to be written. See Figure 5.

The magnitude of the size of the DOD Audit Routines was approaching 100,000 lines of source coding, making up 130 programs. The number of environmental changes (resolution of implementor-names) was in the neighborhood of 1,000 and the number of operating system control cards required to execute the program would be from 1,300 to 5,000 depending on the complexity of the operating system involved.

This was where the support program could save a large amount of both work and mistakes. The Versatile Program Management System (VPMS1) was designed to handle all of these problems with a minimum of effort.

Versatile program management system (VPMS1)

A good portion of the merging included additional enhancements to the VPMS1 (support program) which, by this time, through an evolutionary process had learned to manage two new languages; FORTRAN and JOVIAL. The program had been modified based on the additional requirements of various operating systems for handling particular COBOL problems; the need for making the system easy for the user to interface with, and the need to provide all interfaces between the user, the audit routines, and the operating system.

The introduction of implementor names through "X-cards"

The first problem was the resolution of implementor-names within the source COBOL programs making up the audit routines. In the COBOL language, particularly in the Environment Division, there are constructs which must contain an implementor-defined word in order for the statement to be syntactically complete. Figure 6 shows where the implementor-names must be provided.

THE NOTE placed as the first word in the paragraph causes the entire paragraph to be treated as comments. Instead of the "GO TO ADD-WRITE-1" statement being executed, the logic of the program falls into the delete paragraph which causes the output results to reflect the fact that the test was deleted.

If the syntax error is in the Data Division, then the coding itself must be modified. VPMS1 shows, in its own printed output, the old card image as well as the new card image so that what has been altered is readily apparent, i.e.,

```
012900 02 A PIC ZZ9 Value '1'. NC1085.2 OLD

012900 02 A PIC ZZ9 Value 1.   NC108*RE NEW
```

```
ENVIRONMENT DIVISION.
SOURCE-COMPUTER.
  implementor-name-1.
OBJECT-COMPUTER.
  implementor-name-2.
SPECIAL-NAMES.
  implementor-name-3 is MNEMONIC-NAME
                    .
                    .
                    .
FILE-CONTROL
  SELECT FILE-NAME ASSIGN TO implementor-name-4.
                    .
                    .
                    .
                    .
data division.

  FD FILE-NAME
    VALUE OF implementor-name-5 IS implementor-defined.
```

Figure 6—Implementor defined names that would appear in a COBOL program

If, while executing the object program of an audit routine, an abnormal termination occurs, then a change is required. The cause might be, for example, a data exception or a program loop due to the incorrect implementation of a COBOL statement. In any case, the test in question would have to be deleted. The NOTE would be used as specified above.

In addition, VPMS1 provides a universal method of updating source programs so that the individual who validates more than one compiler is not constantly required to learn new implementor techniques for updating source programs.

Example of update cards through VPMS1:

```
012900  02 A PIC ZZ9     (If the sequence number is
           VALUE 1.        equal the card is replaced;
013210  MOVE 1 TO A.       if there is no match the
014310  NOTE               card is inserted in the ap-
                           propriate place in the
                           program.)
014900*           (Deletes card 014900)
029300*099000     (Deletes  the  series  from  029300
                    through 099000).
```

To carry the problem a step further. Some of the names used by different implementors for the high speed printer in the SELECT statement have been PRINTER, SYSTEM-PRINTER, FORM-PRINTER, SYSOUT, SYSOU1, P1 FOR LISTING, ETC. It is obvious to a programmer what the implementor has in mind, but the compiler that expects SYSTEM-PRINTER, will certainly reject any of the other names. Therefore, each occurrence of an implementor-name must be converted to the correct name. The approach taken is that each implementor-name is defined to VPMS1. For example, the printer is known as XXXX36 and the audit routines using the printer would be set up in the following way:

```
SELECT PRINT-FILE ASSIGN TO
  XXXXX36
```

And the user would provide the name to be used by the computer being tested through an "X-CARD."

```
X-36 SYSTEM-PRINTER
```

VPMS1 would then replace all references of XXXXX36 with SYSTEM-PRINTER.

```
SELECT PRINT-FILE ASSIGN TO
SYSTEM-PRINTER.
```

Ability to update programs

The next problem was to provide the user with a method for making changes to the audit routines in

```
ADD-TEST-1.
   NOTE (Inserted by the user as an update to the program.)
   MOVE 1 TO ALPHA.
   .
   TO TO ADD-WRITE-1.
ADD-DELETE-1.
   PERFORM DELETE.
   .
   .
   .
```

Figure 7—Example of deleting a test in the DOD CCVS

an orderly fashion and at the same time provide a maximum amount of documentation for each change made. There are two reasons for the user to need to make modifications to the actual audit routines:

a. If the compiler will not accept a form of syntax it must be eliminated in order to create a syntactically correct program. There are two ways to accomplish this. In the Procedure Division the NOTE statement is used to force the "invalid" statements to become comments. The results of this action would cause the test to be deleted and this would be reflected in the output. See Figure 7.

OPERATING SYSTEM CONTROL CARD GENERATION

The third problem was the generation of operating system control cards in the appropriate position relative to the source programs in order for the programs to be compiled, loaded and executed. This was the biggest challenge for VPMS1; a COBOL program which had to be structurally compatible with all COBOL compilers and which also had to be able to interface with all operating systems with a negligible amount of modification for each system.

The philosophy of the output of VPMS1 is a file acceptable to a particular operating system as input. For the most part this file closely resembles what would normally be introduced to the operating system through the system's input device or card reader, i.e., control cards, source program, data, etc.

The generation of operating system control cards is based on the specific placement of the statement and the requirement or need for specific statements to accomplish additional functions. These control cards are presented to VPMS1 in a form that will not be intercepted by the operating system and are annotated as to their appropriate characteristics. The body of the actual control card starts in position 8 of the input record. Position one is reserved for a code that specifies the type of control card. The following is allowed in specifying control cards: Initial control cards are generated once at the beginning of the file. Beginning control cards are generated before each source program with a provision for specifying control cards which are generated at specific times, i.e., JOB type cards, subroutine type cards, library control cards, etc. Ending control cards are generated after each source program with the same provision as beginning control cards. Terminal control cards are generated prior to the file being closed. Additional control cards are generated for assigning hardware devices to the object program, bracketing data and for assigning work areas to be used by the COBOL Sort.

There are approximately 25 files used by the entire set of validation routines for which control cards may need to be prepared. In addition to the control cards and information for the Environment Division, the total number of control statements printed for VPMS1 could be in the neighborhood of 200 card images and the possible number of generated control cards on the output file could be as large as 5000. The saving in time and JCL errors that could be prevented should be obvious at this point.

This Environmental information need not be provided by the user because once a set of VPMS1 control cards has been satisfactorily debugged on the system in question, they can be placed in the library file that contains the same program so that a single request could extract the VPMS1 control cards for a given system.

CONCLUSION

It has been demonstrated that the validation of COBOL compilers is possible and that the end result is beneficial to both compiler writers and the users of these compilers. The ease with which the DOD CCVS can be automatically adapted to a given computer system has eliminated approximately 85 to 90 percent of the work involved in validating a COBOL compiler.

Although most compilers are written from the same basic specifications (i.e., the American National Standard COBOL, X3.23-1968, or the CODASYL COBOL Journal of Development) the results are not always the same. The DOD CCVS has exposed numerous compiler bugs as well as misinterpretations of the language. Due to this and similar efforts in the area of compiler validation, the compatibility of today's compilers has grown to a high degree.

We are now awaiting the next version of the American National Standard COBOL. The new specifications will provide an increased level of compatibility between compilers because the specifications are more definitive and contain fewer "implementor defined" areas. In addition, numerous enhancements and several clarifications have been included in the new specification—all contributing to better software, both at the compiler and the application level.

REFERENCES

1 American National Standard COBOL X3.23-1968
American National Standards Institute Inc. New York 1968

2 COBOL-61 Conference on Data System Languages
U. S. Government Printing Office Washington D. C. 1961

A prototype automatic program testing tool

by LEON G. STUCKI

McDonnell Douglas Astronautics Company
Huntington Beach, California

> *". . . as a slow-witted human being I have a very small head and had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure."*
>
> —*Edsger W. Dijkstra*

SOFTWARE SYSTEMS MEASUREMENT TECHNIQUES

The measurement process plays a vital role in the quality assurance and testing of new hardware systems. To insure the reliability of the final hardware system, each stage of development incorporates performance standards and testing procedures. The establishment of software performance criteria has been very nebulous. At first the desire to "just get it working" prevailed in most software development efforts. With the increasing complexity of new and evolving software systems, improved measurement techniques are needed to facilitate disciplined program testing beyond merely debugging. The Program Testing Translator is an automatic tool designed to aid in the measurement and testing of software systems.

A great need exists for new methods of gaining insight into the structure and behavior of programs being tested. Dijkstra alludes to this in a hardware analogy. He points out that the number of different multiplications possible with a 27-bit fixed-point multiplier is approximately 2^{54}. With a speed in the order of tens of microseconds, what at first might seem reasonable to verify, would require 10,000 years of computation.[1] With these possibilities for such a simple operation as the multiplication of two data items, can it be expected that a programmer anticipate completely the actions of a large program?

Dijkstra, in relation to both hardware and software "mechanisms," continues by stating:

> *"The straightforward conclusion is the following: a convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box. I shall call this 'taking the structure of the mechanism into account.' "*[1]

As suggested by R. W. Bemer and A. L. Ellison in their 1968 IFIP report, the examination of hardware and software structures might incorporate similar test procedures:

> *"Instrumentation should be applied to software with the same frequency and unconscious habit with which oscilloscopes are used by the hardware engineer."*[2]

Early attempts at the application of measurement techniques to software dealt mainly with efforts to measure the hardware utilization characteristics. In an attempt to further improve hardware utilization, several aids have been developed ranging from optimized compilers to automated execution monitoring systems.[3,4] The Program Testing Translator, designed to aid in the testing of programs, goes further. In addition to providing execution time statistics on the frequency of execution for various program statements, the Program Testing Translator performs a "standards" check to insure programmers' compliance to an established coding standard, gathers data on the extent to which various branches of a program are executed, and provides data range values on assignment statements and DO-loop control variables.

As was pointed out by Heisenberg, in reference to the measurement of physical systems, a degree of uncertainty is introduced into any system under observation. With Heisenberg's Uncertainty Principle in mind, the Program Testing Translator is presented as a "tool" to be used in the software measurement process. Just as using a microscope to determine the position of a free particle introduces a degree of uncertainty into observations,[5] so must it be concluded that no program measurement tool can guarantee the complete absence of all possible side effects. In particular, potential problems involving changes in time and space must be considered. For example, the behavior of some real-time applications may be affected by increased execution times. To avoid the use and development of more powerful program testing tools because of possible uncertainties, however, would be as great a folly as to reject the use of the microscope.

DATA ANALYZED BY THE PROGRAM TESTING TRANSLATOR

In a paper by Knuth a large sample of FORTRAN programs was quantitatively analyzed in an attempt to come up with a program profile. This profile was expressed in the form of a table of frequency counts showing how often each type of source statement occurs in a "typical" program. Knuth builds a strong case for designing profile-keeping mechanisms into major computer systems.[6] Internal organization of the Program Testing Translator was designed with Knuth's table of frequency counts in mind.

The Program Testing Translator gathers and analyzes data in two general areas: (1) the syntactic profile of the source program showing the number of executable, nonexecutable,* and comment statements, the number of CALL statements and total program branches,** and the number of coding standard's violations, and (2) actual program performance statistics corresponding to various test data sets.

With all options enabled, the actual program performance statistics produced by the Program Testing Translator include:

(1) The number and percentage of those executable source statements actually executed.

(2) The number and percentage of those branches and CALLs actually taken or executed.

(3) The following specific data associated with each executable source statement.
   (a) detailed execution counts
   (b) detailed branch counts on all IF and GOTO statements
   (c) min/max data range values on assignment statements and DO-loop control variables.

Several previous programs[7,8,9] have provided interesting source statement execution data. The additional data range information provided by the Program Testing Translator, however, proves useful in further analyzing program behavior. Extended research investigating possible techniques for automatic test data generation will make use of these data range values. The long term goal of this research is directed toward designing a procedure for obtaining a minimal yet adequate set of test cases for "testing" a program.

STANDARDS' CHECKING

Although general in design, the initial implementation of the Program Testing Translator was restricted to the CDC 6500. Scanning the input source code, the Program Testing Translator flags as warnings all dialect peculiar statements which pose possible machine conversion problems. The standard is basically the ASA FORTRAN IV Standard[10] with some additional restrictions local to McDonnell Douglas. The standard can easily be altered to reflect the needs of a particular installation, in contrast to previous compilers which have incorporated a fixed standard's check (e.g., WATFOR).

DEVELOPMENT OF THE PROGRAM TESTING TRANSLATOR

The Program Testing Translator serves as a prototype automatic testing aid suggesting future development of much more powerful software testing systems. The basic components of the system are the FORTRAN-to-FORTRAN preprocessor and postprocessor module. (See the section on Use of the Program Testing Translator.)

A machine independent Meta Translator[11] was used to generate the Program Testing Translator. Conventionally, moving major software processors between machines posed serious problems requiring either completely new coded versions, or the use of new meta-compiler systems.[12] This Meta Translator produces an

* Executable statements include assignment, control, and input/output statements. Nonexecutable statements include specification and subprogram statements.

** A branch will denote each possible path of program flow in all conditional and transfer statements (i.e., all IF and GOTO statements in FORTRAN).

ASA Standard FORTRAN translator which represents an easily movable translation package.

In general, for implementation on another machine, FORTRAN-to-FORTRAN processors such as the Program Testing Translator require only that the syntactic definition input to the Meta Translator be changed to reflect the syntax of the new machine's FORTRAN dialect.

INSTRUMENTATION TECHNIQUES

The instrumentation technique used by the Program Testing Translator is to insert appropriate high-level language statements within the original source code making as few changes as possible.[13] Three memory areas are added to each main program and subroutine. One is used for various execution counts while the other two are used for the storage of minimum and maximum data range values for specified assignment statements and DO-loop control variables. The size of these respective memory areas depends upon the size of the program being tested and the options chosen.

Simple counters of the form:

$$\text{QINT}(i) = \text{QINT}(i) + 1$$

are inserted at all points prefixed by statement numbers, at entry points, after CALL statements, after logical IF statements, after DO statements, and after the last statement in a DO-loop.[8,14] Additional counters are used to maintain branch counts on all IF and GOTO statements.

Minimum and maximum data range values are calculated following each non-trivial assignment statement. These values of differing types are packed into the two memory areas allocated for this purpose. Minimum and maximum values may also be kept on all variables used as DO-loop control parameters. These values are calculated before entry into the DO-loop.

USE OF THE PROGRAM TESTING TRANSLATOR

Overall program flow of the Program Testing Translator is diagrammed in Figure 1. Basically, the user's FORTRAN source cards are input to the preprocessor. This preprocessor module outputs: (1) an instrumented source file to be compiled by the standard FORTRAN compiler and (2) an intermediate data file for postprocessing. This intermediate file contains a copy of the original source code with a linkage field for extracting the profile and execution-time data for the program.

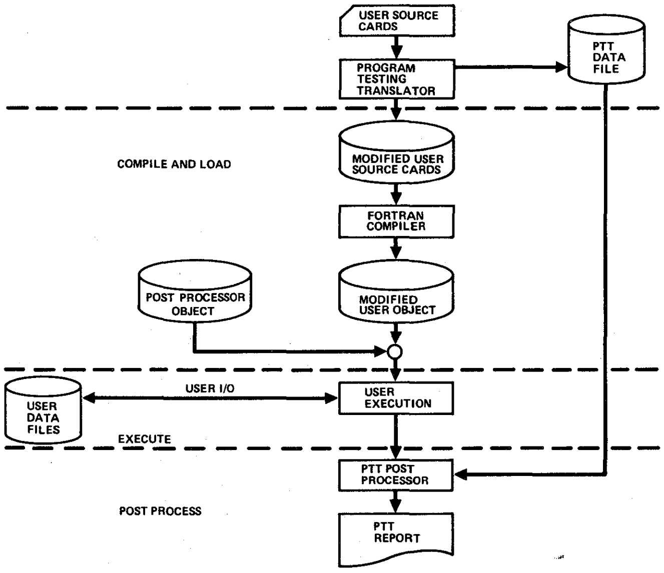


Figure 1—Program testing translation job flow

The object code produced by the FORTRAN compiler is linked with the postprocessor from an object library. The resulting object module can now be saved along with the intermediate Program Testing Translator data file. Together they can then be executed with any number of user test cases. Using the intermediate file built by the preprocessor and data gathered while duplicating the original program results, the postprocessor generates reports showing program behavior for each specific test case. Analysis of these reports will help eliminate redundant test cases and point to sections of the user's program which have not yet been "tested." Examination of these particular areas may lead to either their elimination or the inclusion of modified test cases to check out these program sections.

Preliminary measurements indicate that the execution time of the instrumented object module, with all options enabled, is approximately one and one half to two times the normal CPU time. Increases in I/O time are negligible in most cases.

ACTUAL TEST EXPERIENCE

Although the Program Testing Translator has only been available for a short time, several interesting results have come to light.

One of the first major subroutines, processed at McDonnell Douglas, was an eigenvalue-eigenvector subroutine believed to be the most efficient algorithm currently available for symmetric matrices.[15,16] Of the

613 source statements in the subroutine, it was immediately noted that the nested DO-loop shown here was accounting for one quarter of all the execution counts for the entire subroutine (see Appendix).

```
        DO 640 I=1, N
              ⋮
        DO 640 J=1, N
    640 B(I)=B(I)+A(I, J)*V(J, IVEC)
```

Several immediate observations are worthy of mention. First, note that the complexity of the above statement with its double subscripting and multiplication makes it a costly statement to execute. Second, it can be seen that the subscripting of the variable B(I) could be promoted out of the inner loop. A good optimizing compiler should promote the subscripting of the B(I)'s and produce good code for the double subscripting and multiplication[17] but it cannot logically redesign the nested loop. A redesigned machine language subroutine replacing the original loop has now cut total subroutine execution time by one third. Further analysis of the same program, in an attempt to determine why several sections of code were not executed, revealed a logic error making it impossible to ever reach one particular section of code. This error, which was subsequently corrected, can be seen on the first page of the original run contained in the Appendix. This was a subroutine experiencing a great deal of use and thought to be thoroughly checked out.

Running the Program Testing Translator through itself has resulted in savings of over 37 percent in CPU execution times. The standard's checking performed by the Program Testing Translator has verified the machine independence of the Meta Translator.

Table I contains a summary of the actual program statistics observed on the first eight major programs run through the Program Testing Translator. It is interesting to note that only 45.9 percent of the possible executable statements were actually executed. Of more importance, however, is the fact that only 32.5 percent of all possible branches were actually taken.

Table II compares the class profile data gathered at McDonnell Douglas by the Program Testing Translator with the Lockheed and Stanford findings cited by Knuth.[6] The syntactic profile of the McDonnell Douglas and Lockheed samples were remarkably similar. Stanford's "student" job profile shows much less

TABLE I—Actual Program Statistics with the Program Testing Translator

Program	AB33	AD77	F999	JOYCE	META	MI01	PTT	UT03	TOTALS
Total Number of Statements	1,578	11,111	2,833	3,033	1,125	775	772	1,445	22,672
No. of Comment Statements	355	3,847	644	176	86	189	44	54	5,395
Percentage of Total	22.5	34.6	22.7	5.8	7.6	24.4	5.7	3.7	23.8
No. Other Nonexecutable Statements	177	905	257	372	534	40	249	254	2,788
Percentage of Total	11.2	8.1	9.1	12.3	47.5	5.2	32.3	17.6	12.3
No. Standard's Violations	9	33	28	65	1	1	23	44	204
Percentage of Total	0.6	0.3	1.0	2.1	0.1	0.1	3.0	3.0	1.0
No. Executable Statements	1,046	6,359	1,932	2,485	505	546	479	1,137	14,489
Percentage of Total	66.3	57.2	68.2	81.9	44.9	70.5	62.0	78.7	63.9
No. Actually Executed	678	2,213	1,155	846	419	392	364	584	6,651
Percentage Executed	64.8	34.8	59.8	34.0	83.0	71.8	76.0	51.4	45.9
No. of Branches	357	2,635	859	1,718	355	189	333	510	6,956
Avg./Exec. Statements	0.34	0.41	0.44	0.69	0.70	0.35	0.70	0.45	0.48
No. Actually Executed	195	571	376	454	203	112	175	175	2,261
Percentage Executed	54.6	21.7	43.8	26.4	57.2	59.3	52.6	34.3	32.5
No. of CALL Statements	20	369	86	278	32	9	19	99	912
Avg./Exec. Statements	0.02	0.06	0.04	0.11	0.06	0.02	0.04	0.09	0.06
No. Actually Executed	18	119	26	67	21	3	5	76	335
Percentage Executed	90.0	32.2	30.2	24.1	65.6	33.3	26.3	76.8	36.7
Total Statement Exec. Counts (in thousands)	26,772	2,929	112	1,129	5,284	1,133	1,087	71	38,517

TABLE II—A Comparison of Syntatic Class Profiles

	McDonnell Douglas	Lockheed*	Stanford*
Total No. Statements	22,672	245,000	10,700
Percentage Comments	23.8	21.6	10.2
Percentage Other Nonexecutable	12.3	10.6**	12.3**
Percentage Executable	63.9	67.8	77.5
Avg. No. Branches/ Executable Statement	0.48	0.54	0.32
Avg. No. CALLs/ Executable Statement	0.06	0.09	0.04

* Note: These figures represent this author's best attempt at extrapolating comparable measurements from Knuth's paper.[6] Knuth's percentage figures had to be corrected by adding the comment statements into the total number of statements. Calculations of the average number of branches per executable statement require two assumptions: (1) 30 percent of the IF statements had 3 possible branches while 70 percent had 2 branches, (2) 96 percent of the GOTO statements were unconditional (i.e., 1 branch), while 4 percent were switched (i.e., 2 branches were assumed).

** Includes the following: FORMAT, DATA, DIMENSION, COMMON, END, SUBROUTINE, EQUIVALENCE, INTEGER, ENTRY, LOGICAL, REAL, DOUBLE, OVERLAY, EXTERNAL, IMPLICIT, COMPLEX, NAMELIST, BLOCKDATA.

emphasis on internal documentation (i.e., fewer comment statements) and also exemplifies a more straightforward approach to flow of control (as seen in Stanford's 0.32 branches per executable statement compared to 0.48 and 0.54 branches per executable statement for the two aerospace companies).

EXTENSIONS OF THE PROGRAM TESTING TRANSLATOR

As alluded to in earlier sections, much more powerful testing systems can and should be built in the future. Relatively simple changes to the postprocessor module could enable the execution time data from multiple test runs to be combined automatically into composite test reports.

Changes to the translator module might provide the options of first and last values on assignment statements as well as range values.

Modeled after development of the FORTRAN-to-FORTRAN system, instrumentation systems for other languages of heavy use such as COBOL or PL/1 might well be developed.

The most important area now being investigated, however, is the possible design of extensible automatic testing aids to provide for the automatic generation of test data. Evolvement of future testing tools along these lines would greatly aid the quality assurance aspects of large software systems.

ACKNOWLEDGMENT

The research described in this report was carried out by the author under the direction of T. W. Miller, Jr. and R. G. Koppang in the Advance Computer Systems Department at the McDonnell Douglas Astronautics Company in Huntington Beach, California.

REFERENCES

1 E W DIJKSTRA
Notes on structured programming
Technological University Eindhaven The Netherlands Department of Mathematics April 1970 TH Report 70-WSK-03

2 R W BEMER A L ELLISON
Software instrumentation systems for optimum performance
IFIP Congress 1968 Proceedings Software Session 2 Booklet C p 39-42

3 *System measurement software SMS/360 problem program efficiency*
Boole and Babbage Inc. Product Description Palo Alto California May 1969 Document No. S-32 Rev-1

4 D N KELLY
SPY a computer program execution monitoring package
McDonnell Douglas Automation Company Huntington Beach California MDC G2659 December 1971

5 W HEISENBERG
The uncertainty principle
Zeitschrift fuer Physic Vol 43 1927

6 D E KNUTH
An empirical study of FORTRAN programs
Stanford Artificial Intelligence Project Memo AIM-137
Computer Science Dept Report No. CS-186

7 1ST LT G W JOSEPH
The fortran frequency analyzer as a data gathering aid for computer system simulation
Electronics Systems Division United States Air Force
L G Hanscom Field Bedford Massachusetts March 1972

8 D H H IGNALLS
FETE A FORTRAN execution time estimator
Computer Science Department Stanford University
STAN-CS-71-204 February 1971

9 *CDC 6500 FLOW user's manual*
TRW Systems Group Redondo Beach California

10 *Proposed American standard X3-4.3-FORTRAN*
Inquiries addressed to X3 Secretary BEMA
235 E 42nd Street New York NY March 1965

11 *Meta translator*
Advanced Computer Sciences Department McDonnell Douglas Astronautics Company Huntington Beach California currently in preparation

12 A R TYRILL
The meta 7 translator writing system
Master's Thesis School of Engineering and Applied Science University of California Los Angeles California Report 71-22 September 1971

13 E C RUSSELL
Automatic program analysis
PhD Dissertation in Engineering School of Engineering and Applied Science University of California Los Angeles California 1969

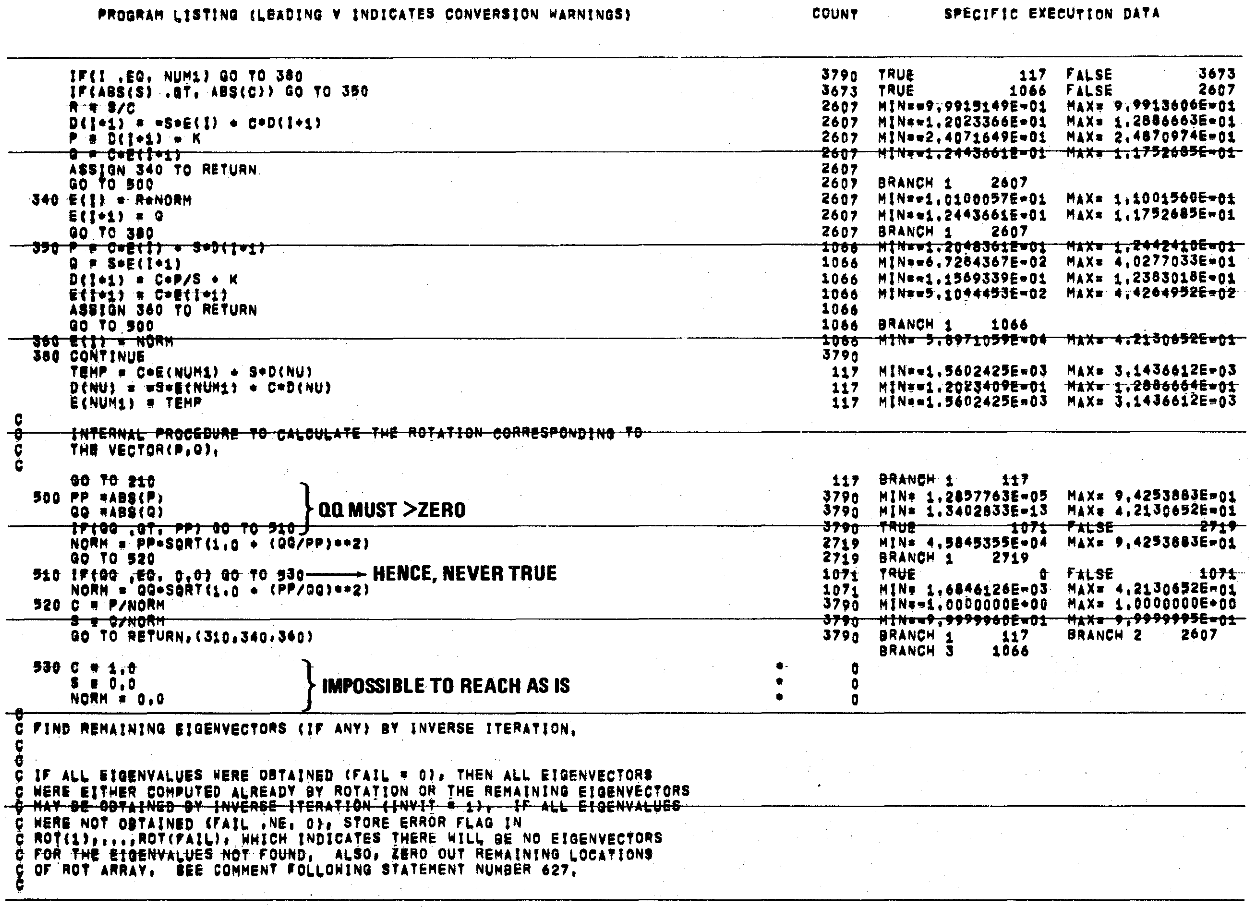
14 V G CERF
Measurement of recursive programs
Master's Thesis School of Engineering and Applied Science University of California Los Angeles California 70-43 May 1970

15 S J CLARK
Computation of eigenvalues and eigenvectors of a real symmetric matrix using SYMQR1
Advanced Mathematics Department McDonnell Douglas Astronautics Company Huntington Beach California Internal Memorandum A3-830-BEGO-71-07 November 1971

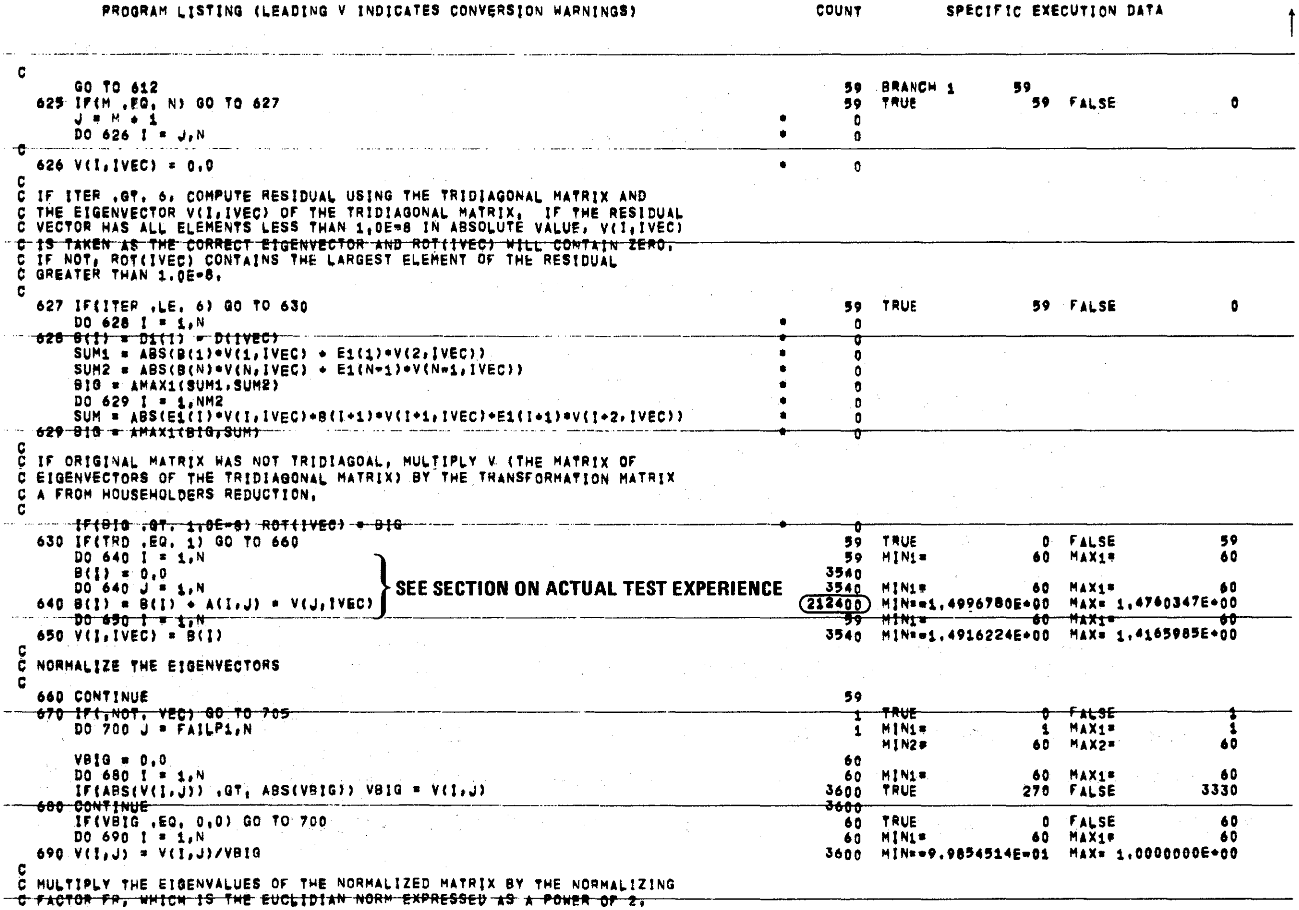
16 S J CLARK
Further improvement of subroutine SYMQR1
Advanced Mathematics Department McDonnell Douglas Astronautics Company Hungtington Beach California Internal Memorandum A3-830-BEGO-SJC-094 March 1972

17 F E ALLEN
Program optimization
Ann Rev in Automatic Programming 5(1969) pp 237-307

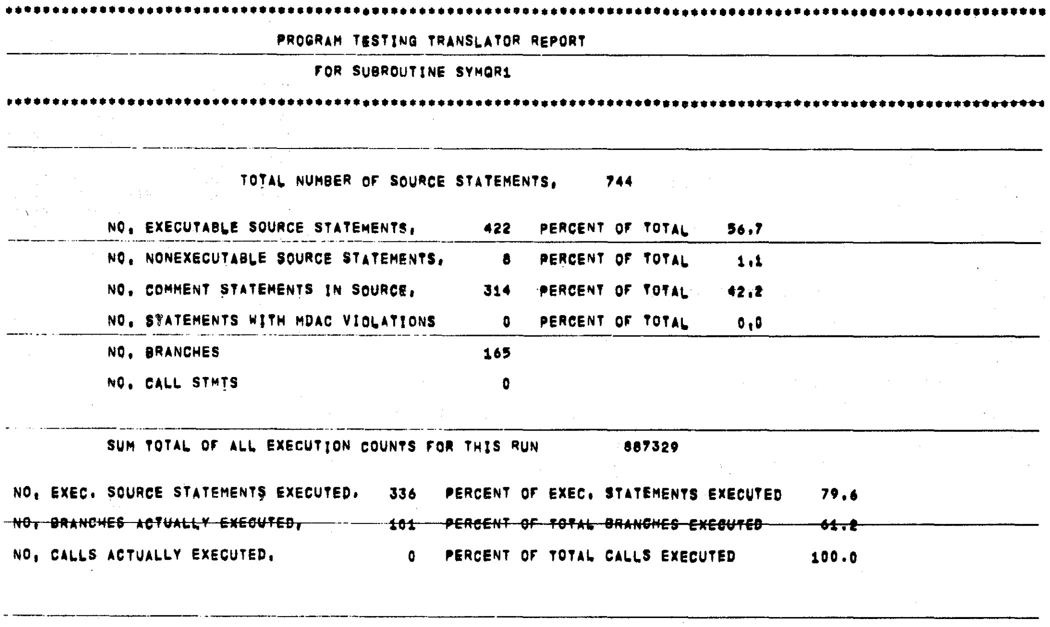
APPENDIX



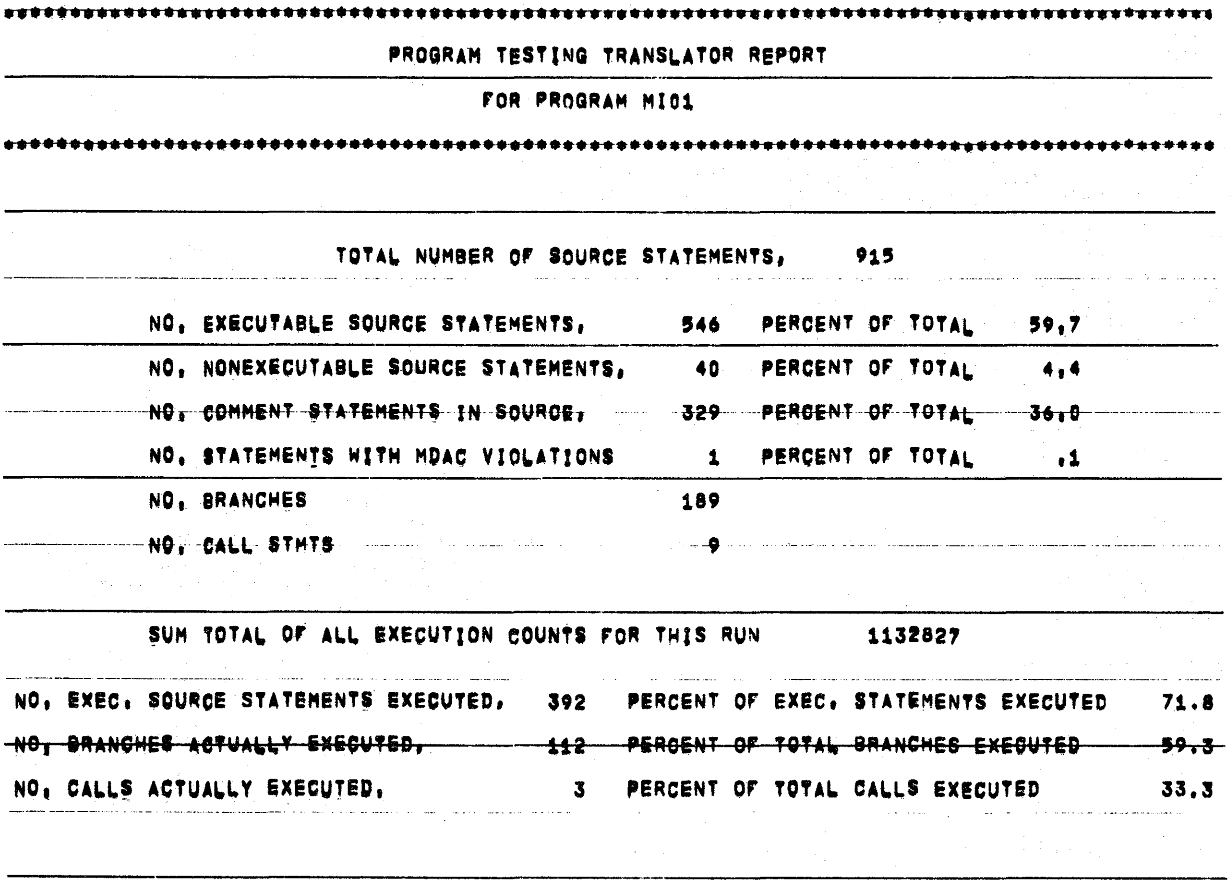
```
          PROGRAM LISTING (LEADING V INDICATES CONVERSION WARNINGS)                    COUNT         SPECIFIC EXECUTION DATA

      IF(I .EQ. NUM1) GO TO 380                                                       3790  TRUE          117  FALSE         3673
      IF(ABS(S) .GT. ABS(C)) GO TO 350                                                3673  TRUE         1066  FALSE         2607
      R = S/C                                                                         2607  MIN=-9.9915149E-01  MAX= 9.9913606E-01
      D(I+1) = -S*E(I) + C*D(I+1)                                                     2607  MIN=-1.2023366E-01  MAX= 1.2886663E-01
      P = D(I+1) - K                                                                  2607  MIN=-2.4071649E-01  MAX= 2.4870974E-01
      Q = C*E(I+1)                                                                    2607  MIN=-1.2443661E-01  MAX= 1.1752685E-01
      ASSIGN 340 TO RETURN                                                            2607
      GO TO 500                                                                       2607  BRANCH 1     2607
  340 E(I) = R*NORM                                                                   2607  MIN=-1.0100057E-01  MAX= 1.1001560E-01
      E(I+1) = Q                                                                      2607  MIN=-1.2443661E-01  MAX= 1.1752685E-01
      GO TO 380                                                                       2607  BRANCH 1     2607
  350 P = C*E(I) + S*D(I+1)                                                           1066  MIN=-1.2048361E-01  MAX= 1.2442410E-01
      Q = S*E(I+1)                                                                    1066  MIN=-6.7264367E-02  MAX= 4.0277033E-01
      D(I+1) = C*P/S + K                                                              1066  MIN=-1.1569339E-01  MAX= 1.2383018E-01
      E(I+1) = C*E(I+1)                                                               1066  MIN=-5.1044453E-02  MAX= 4.4264952E-02
      ASSIGN 360 TO RETURN                                                            1066
      GO TO 500                                                                       1066  BRANCH 1     1066
  360 E(I) = NORM                                                                     1066  MIN= 5.8971059E-04  MAX= 4.2130652E-01
  380 CONTINUE                                                                        3790
      TEMP = C*E(NUM1) + S*D(NU)                                                       117  MIN=-1.5602425E-03  MAX= 3.1436612E-03
      D(NU) = -S*E(NUM1) + C*D(NU)                                                     117  MIN=-1.2023409E-01  MAX= 1.2886664E-01
      E(NUM1) = TEMP                                                                   117  MIN=-1.5602425E-03  MAX= 3.1436612E-03
C
C     INTERNAL PROCEDURE TO CALCULATE THE ROTATION CORRESPONDING TO
C     THE VECTOR(P,Q).
C
      GO TO 210                                                                        117  BRANCH 1      117
  500 PP =ABS(P)          ⎫                                                           3790  MIN= 1.2857763E-05  MAX= 9.4253883E-01
      QQ =ABS(Q)          ⎬ QQ MUST >ZERO                                             3790  MIN= 1.3402833E-13  MAX= 4.2130652E-01
      IF(QQ .GT. PP) GO TO 510 ⎭                                                      3790  TRUE         1071  FALSE         2719
      NORM = PP*SQRT(1.0 + (QQ/PP)**2)                                                2719  MIN= 4.5845355E-04  MAX= 9.4253883E-01
      GO TO 520                                                                       2719  BRANCH 1     2719
  510 IF(QQ .EQ. 0.0) GO TO 530 ———→ HENCE, NEVER TRUE                                1071  TRUE            0  FALSE         1071
      NORM = QQ*SQRT(1.0 + (PP/QQ)**2)                                                1071  MIN= 1.6846126E-03  MAX= 4.2130652E-01
  520 C = P/NORM                                                                      3790  MIN=-1.0000000E+00  MAX= 1.0000000E+00
      S = Q/NORM                                                                      3790  MIN=-9.9999960E-01  MAX= 9.9999995E-01
      GO TO RETURN,(310,340,360)                                                      3790  BRANCH 1      117  BRANCH 2     2607
                                                                                            BRANCH 3     1066
  530 C = 1.0             ⎫                                                   *          0
      S = 0.0             ⎬ IMPOSSIBLE TO REACH AS IS                         *          0
      NORM = 0.0          ⎭                                                   *          0
C
C FIND REMAINING EIGENVECTORS (IF ANY) BY INVERSE ITERATION.
C
C
C IF ALL EIGENVALUES WERE OBTAINED (FAIL = 0), THEN ALL EIGENVECTORS
C WERE EITHER COMPUTED ALREADY BY ROTATION OR THE REMAINING EIGENVECTORS
C MAY BE OBTAINED BY INVERSE ITERATION (INVIT = 1).  IF ALL EIGENVALUES
C WERE NOT OBTAINED (FAIL .NE. 0), STORE ERROR FLAG IN
C ROT(1),...,ROT(FAIL), WHICH INDICATES THERE WILL BE NO EIGENVECTORS
C FOR THE EIGENVALUES NOT FOUND.  ALSO, ZERO OUT REMAINING LOCATIONS
C OF ROT ARRAY.  SEE COMMENT FOLLOWING STATEMENT NUMBER 627.
C
```



```
        PROGRAM LISTING (LEADING V INDICATES CONVERSION WARNINGS)              COUNT       SPECIFIC EXECUTION DATA

C
     GO TO 612                                                                  59  BRANCH 1        59
 625 IF(M .EQ. N) GO TO 627                                                     59  TRUE            59  FALSE           0
     J = M + 1                                                           *       0
     DO 626 I = J,N                                                      *       0
C
 626 V(I,IVEC) = 0.0                                                     *       0
C
C IF ITER .GT. 6, COMPUTE RESIDUAL USING THE TRIDIAGONAL MATRIX AND
C THE EIGENVECTOR V(I,IVEC) OF THE TRIDIAGONAL MATRIX.  IF THE RESIDUAL
C VECTOR HAS ALL ELEMENTS LESS THAN 1.0E-8 IN ABSOLUTE VALUE, V(I,IVEC)
C IS TAKEN AS THE CORRECT EIGENVECTOR AND ROT(IVEC) WILL CONTAIN ZERO.
C IF NOT, ROT(IVEC) CONTAINS THE LARGEST ELEMENT OF THE RESIDUAL
C GREATER THAN 1.0E-8.
C
 627 IF(ITER .LE. 6) GO TO 630                                                  59  TRUE            59  FALSE           0
     DO 628 I = 1,N                                                      *       0
 628 B(I) = D1(I) - D(IVEC)                                              *       0
     SUM1 = ABS(B(1)*V(1,IVEC) + E1(1)*V(2,IVEC))                        *       0
     SUM2 = ABS(B(N)*V(N,IVEC) + E1(N-1)*V(N-1,IVEC))                    *       0
     BIG = AMAX1(SUM1,SUM2)                                              *       0
     DO 629 I = 1,NM2                                                    *       0
     SUM = ABS(E1(I)*V(I,IVEC)+B(I+1)*V(I+1,IVEC)+E1(I+1)*V(I+2,IVEC))   *       0
 629 BIG = AMAX1(BIG,SUM)                                                *       0
C
C IF ORIGINAL MATRIX WAS NOT TRIDIAGOAL, MULTIPLY V (THE MATRIX OF
C EIGENVECTORS OF THE TRIDIAGONAL MATRIX) BY THE TRANSFORMATION MATRIX
C A FROM HOUSEHOLDERS REDUCTION.
C
     IF(BIG .GT. 1.0E-8) ROT(IVEC) = BIG                                 *       0
 630 IF(TRD .EQ. 1) GO TO 660                                                   59  TRUE             0  FALSE          59
     DO 640 I = 1,N                          )                                  59  MIN1=           60  MAX1=          60
     B(I) = 0.0                              |                                3540
     DO 640 J = 1,N                          } SEE SECTION ON ACTUAL TEST EXPERIENCE
                                             |                                3540  MIN1=           60  MAX1=          60
 640 B(I) = B(I) + A(I,J) * V(J,IVEC)        )                              (212400) MIN=-1.4996780E+00  MAX= 1.4760347E+00
     DO 650 I = 1,N                                                             59  MIN1=           60  MAX1=          60
 650 V(I,IVEC) = B(I)                                                         3540  MIN=-1.4916224E+00  MAX= 1.4165985E+00
C
C NORMALIZE THE EIGENVECTORS
C
 660 CONTINUE                                                                   59
 670 IF(.NOT. VEC) GO TO 705                                                     1  TRUE             0  FALSE           1
     DO 700 J = FAILP1,N                                                         1  MIN1=            1  MAX1=           1
                                                                                    MIN2=           60  MAX2=          60
     VBIG = 0.0                                                                 60
     DO 680 I = 1,N                                                             60  MIN1=           60  MAX1=          60
     IF(ABS(V(I,J)) .GT. ABS(VBIG)) VBIG = V(I,J)                             3600  TRUE           270  FALSE        3330
 680 CONTINUE                                                                 3600
     IF(VBIG .EQ. 0.0) GO TO 700                                                60  TRUE             0  FALSE          60
     DO 690 I = 1,N                                                             60  MIN1=           60  MAX1=          60
 690 V(I,J) = V(I,J)/VBIG                                                     3600  MIN=-9.9854514E-01  MAX= 1.0000000E+00
C
C MULTIPLY THE EIGENVALUES OF THE NORMALIZED MATRIX BY THE NORMALIZING
C FACTOR FP, WHICH IS THE EUCLIDIAN NORM EXPRESSED AS A POWER OF 2.
```



```
****************************************************************************************************

                              PROGRAM TESTING TRANSLATOR REPORT

                                 FOR SUBROUTINE SYMQR1

****************************************************************************************************


                     TOTAL NUMBER OF SOURCE STATEMENTS,      744

            NO. EXECUTABLE SOURCE STATEMENTS,        422   PERCENT OF TOTAL    56.7

            NO. NONEXECUTABLE SOURCE STATEMENTS,       8   PERCENT OF TOTAL     1.1

            NO. COMMENT STATEMENTS IN SOURCE,        314   PERCENT OF TOTAL    42.2

            NO. STATEMENTS WITH MDAC VIOLATIONS        0   PERCENT OF TOTAL     0.0

            NO. BRANCHES                             165

            NO. CALL STMTS                             0


            SUM TOTAL OF ALL EXECUTION COUNTS FOR THIS RUN          887329

NO. EXEC. SOURCE STATEMENTS EXECUTED.  336   PERCENT OF EXEC. STATEMENTS EXECUTED     79.6

NO. BRANCHES ACTUALLY EXECUTED,        101   PERCENT OF TOTAL BRANCHES EXECUTED       61.2

NO. CALLS ACTUALLY EXECUTED.             0   PERCENT OF TOTAL CALLS EXECUTED         100.0
```



```
***************************************************************************************

                         PROGRAM TESTING TRANSLATOR REPORT

                                FOR PROGRAM MI01

***************************************************************************************


                    TOTAL NUMBER OF SOURCE STATEMENTS,     915

          NO, EXECUTABLE SOURCE STATEMENTS,        546   PERCENT OF TOTAL    59,7

          NO, NONEXECUTABLE SOURCE STATEMENTS,      40   PERCENT OF TOTAL     4,4

          NO, COMMENT STATEMENTS IN SOURCE,        329   PERCENT OF TOTAL    36,0

          NO, STATEMENTS WITH MDAC VIOLATIONS        1   PERCENT OF TOTAL      ,1

          NO, BRANCHES                             189

          NO, CALL STMTS                             9


          SUM TOTAL OF ALL EXECUTION COUNTS FOR THIS RUN        1132827

NO, EXEC. SOURCE STATEMENTS EXECUTED,   392   PERCENT OF EXEC. STATEMENTS EXECUTED    71.8

NO, BRANCHES ACTUALLY EXECUTED,         112   PERCENT OF TOTAL BRANCHES EXECUTED      59.3

NO, CALLS ACTUALLY EXECUTED,              3   PERCENT OF TOTAL CALLS EXECUTED         33.3
```

An approach to software reliability prediction and quality control*

by NORMAN F. SCHNEIDEWIND

Naval Postgraduate School
Monterey, California

INTRODUCTION

The increase in importance of software in command and control and other complex systems has not been accompanied by commensurate progress in the development of analytical techniques for the measurement of software quality and the prediction of software reliability. This paper presents a rationale for implementing software reliability programs; defines software reliability; and describes some of the problems of performing software reliability analysis. A software reliability program is outlined and a methodology for reliability prediction and quality control is presented. The results of initial efforts to develop a software reliability methodology at the Naval Electronics Laboratory Center are reported.

RATIONALE OF SOFTWARE RELIABILITY

The purpose of a software reliability and quality control program is to provide a means for establishing quantitative criteria for the acceptance or rejection of software and to provide a method for predicting the reliability of software under operating conditions. A computer system consists of hardware, software and human operators. Within the software sub-system, there may exist a number of modules or programs. A total reliability analysis would address the reliability requirements of each major sub-system: hardware, software and operators and for each component within a sub-system. Within the hardware sub-system, reliability estimates should be provided for the central processing unit, discs, magnetic tapes, and other peripheral units. Within the software sub-system, reliability estimates should be provided for each module or program.

Relatively little work has been done in the areas of software and human operator reliabilities, despite the fact that these sub-systems are as important as hardware in determining total system reliability. This research effort is directed toward the goals of developing methodologies and programs for software reliability prediction and quality control.

SOFTWARE RELIABILITY PROGRAM

A description of the elements of a software reliability program follows.

Reliability specification

Reliability specifications are established in advance of software production in order to provide quantitative criteria for the acceptance or rejection of software products. Without such a specification there is no objective criteria on which to judge the quality of a program. Software reliability specifications would be determined from an analysis of total computer system reliability requirements. Individual program or module specifications would be determined by allocating to a program the reliability necessary to achieve the desired total computer system reliability, when all hardware, software and operator reliabilities are considered. Prevailing practice is to consider only hardware when establishing reliability specifications. The matter of establishing software reliability specifications has been largely ignored. The reliability program described here would make explicit provision for software reliability.

* This work was supported by the Computer Sciences Department of the Naval Electronics Laboratory Center under Project P509001. The opinions and assertions contained herein are the private ones of the writer and are not to be construed as official, or as reflecting the views of the Department of the Navy or the Naval service at large.

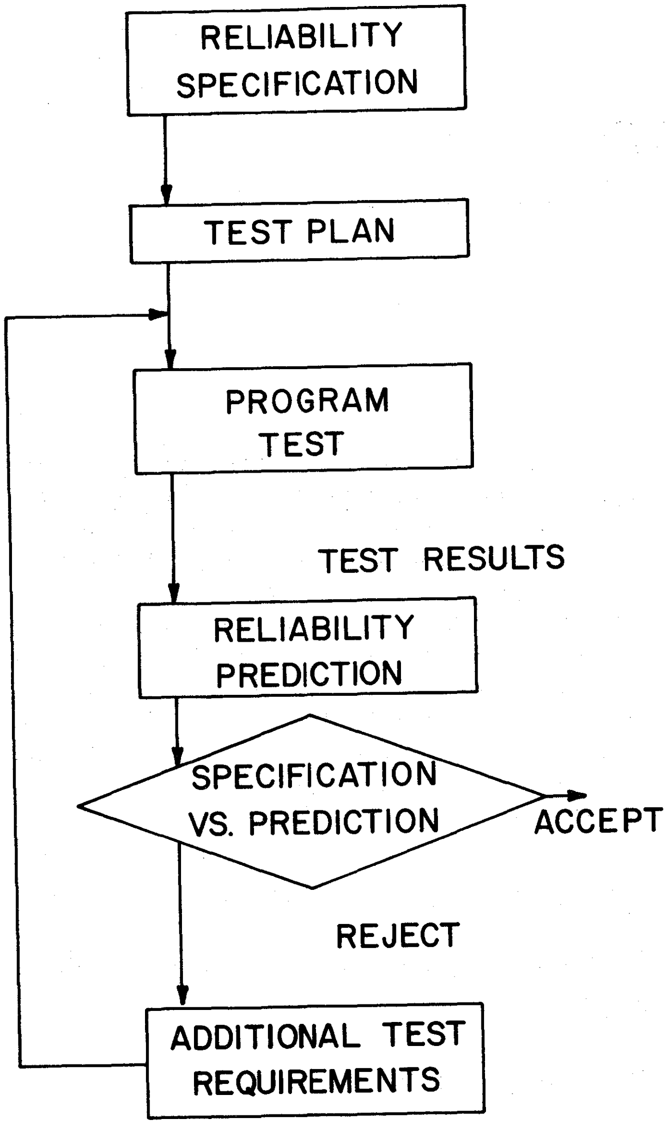

RELIABILITY SPECIFICATION
TEST PLAN
PROGRAM TEST
TEST RESULTS
RELIABILITY PREDICTION
SPECIFICATION VS. PREDICTION
ACCEPT
REJECT
ADDITIONAL TEST REQUIREMENTS

Figure 1—Reliability sequence

Reliability specifications are also used to establish initial test performance requirements in terms of test time and number of troubles. These requirements pertain to the formal test period which starts after the program has been released by the programmer and submitted for independent test. This period also constitutes the reliability demonstration period. During the formal test period, a program must operate for a period of time with less than a specified number of troubles. Satisfying this requirement constitutes meeting the reliability specification. Testing and computation of test requirements proceeds in stages. A stage is a test period during which an attempt is made to qualify a program. The number of stages is governed by the number of test periods which are required to demonstrate reliability.

Reliability prediction

The main purpose of reliability prediction is to provide an estimate of *future* probability of successful program operation. As in any reliability program, this estimate is based on historical and current test results. The reliability prediction is updated with new test data at the end of each stage of testing. In addition to prediction, reliability is used for quality control. Predicted reliability is compared with specified reliability. If a program does not qualify at a given stage, test requirements are computed for the next stage which, if satisfied, will result in the program satisfying the reliability specification. For example, assume on the basis of the reliability specification, that a program must operate for 100 hours during the first stage with no more than one trouble of a specified severity level in order to qualify. Assume that three troubles occur during the first stage; the program fails the first test. The test requirements for stage two which are necessary in order for the predicted reliability to equal or exceed the specified reliability would be determined. In this case, the additional requirement might be to operate another 200 hours without trouble. This process is repeated until the required reliability has been demonstrated. When a program does qualify, the final estimate of predicted reliability (applicable to actual operation) is computed. The process is summarized in Figure 1.

PROBLEMS IN SOFTWARE RELIABILITY ANALYSIS

There are many conceptual and definitional problems associated with software reliability. Some of these problems are described below.

Classification of troubles

Frequently the source of an error—whether it be hardware or software—cannot be definitely established. For example, a memory-to-memory transfer may produce incorrect data at the destination locations. Was this error due to a marginal memory unit or to a defective program? It may require days or weeks before the cause of the error can be positively identified. Another difficulty is that software errors can result from either an operating system or applications programs. It may be difficult to establish whether:

1. The error was actually an error in the application program, or
2. was a violation of operating system protocol (Job Control Language), or
3. was the result of an error in the operating system itself.

Severity and type of trouble

Software troubles must be defined and classified in terms of severity and type. Software troubles differ as to their impact on system operation. A few incorrect characters in textual information which can still be deciphered is much less serious than a transfer outside the memory bounds of a program. Reliability predictions should specify the severity of troubles that are included in the prediction. A reliability predictor may involve one, more than one, or all severity levels. It may be appropriate to have a reliability predictor for each severity level and one which includes all severity levels.

Difference between test and operating environments

It may be difficult to completely duplicate the actual software operating environment during a test. The operating environment may comprise certain inputs, system load or operator actions which cannot be simulated during test. The inability under test conditions to fully duplicate the influence of inputs and stresses placed on the system by uncontrollable external inputs, operator performance, equipment reliability and equipment maintenance practice means that a reliability prediction is only an estimate of the actual reliability which will be obtained in operation. The accuracy of the reliability prediction will improve as the test environment approaches conditions of actual operation.

Adequacy of detail in trouble reporting

It is reasonable to assume that lower levels of program structure will provide greater accuracy of prediction than higher levels. For example, a subroutine may be critical with respect to the operation of a program. Trouble report data at the subroutine level may be more useful than data at the program level. However, in practice, software troubles may not be documented at the level which is most desirable for analysis purposes. Assuming an adequate sample size, the smaller the unit of programming for which error information is obtained, the more accurate the reliability prediction, since many detailed reliability analyses can be combined for the purpose of system reliability estimation. However, a counterbalancing effect is that sample size, in terms of number of troubles, decreases as the unit of programming is decreased. Also, if the unit is too small, the number of program units which must be analyzed in order to compute total system reliability becomes excessive. In practice, the analyst seldom has a choice of levels of program error documentation. The problem is more one of uncovering *any* usable data!

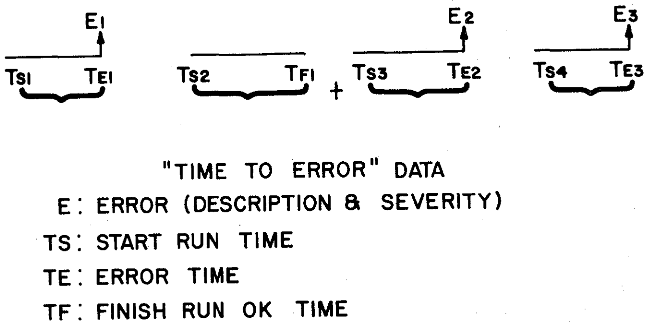


Figure 2—Data collection requirements

Another problem is the absence of data which records the start time of each program test and the times at which trouble occurs, in order that the distribution of time between troubles can be determined. What is required is a time trace of program testing and trouble reporting such as that depicted in Figure 2.

Selection of test sample

Another problem is the possible nonrepresentativeness of trouble report data. The selection of program functions for testing should be proportional to the criticality of a function to mission success and also to the frequency of occurrence of the function during program operation. Frequency of program function testing based on the product of criticality and frequency of use in operation appears to be a reasonable basis for selecting functions for test. To the extent that testing does not occur in accordance with criterion, it will be nonrepresentative of the importance and frequency of use of functions in actual operation, thus causing a bias in reliability prediction. The problem posed by a nonrepresentative sample is difficult to counteract due to the following reasons:

- The selection of the sample is under the control of the test group—not the reliability analyst.

- It may be difficult to obtain information regarding the criticality and frequency of occurrence of program functions.
- Attempts at selecting a representative sample from the available test report data may be infeasible due to existing small samples.

A partial solution to these problems can be obtained by proper design of test procedures or by changes to existing procedures. Software troubles can be classified in detail, test environments can be made realistic to the limit of economic feasibility and representative program functions can be selected for test. The most important and least expensive improvement would be to make test reporting responsive to the needs of reliability analysis. This involves reporting software troubles at least down to the level of the program entity used for reliability prediction and of providing time histories of program troubles. However, even if administrative procedures are changed, the problem of identifying the source of a trouble as to hardware, software or human action will remain. This situation illustrates the need for extensive documentation of program troubles at the time of their occurrence.

DEFINITIONS

Software trouble

In order to provide a method for predicting software reliability, it is first necessary to define software errors or troubles. The following definition will be utilized:

> *A software trouble is any logical or clerical error made by a programmer in creating or coding an algorithm which causes the algorithm to produce an incorrect result when the algorithm is presented with a correct input.*

The above definition excludes errors due to hardware, input or operator action. In addition, it will be understood to exclude compilation errors and errors caused by the operating system. In short, the errors considered in this study are application program errors.

Software reliability

Software reliability $R(t)$ is the probability that a program will operate without a single occurrence of a specified severity of trouble, or worse, for a specified length of time t, and with a specified input load. This is equivalent to the probability that a program will operate successfully for at least time t.

Software probability density function

A probability law $f(t)$ which governs the occurrence of troubles in the operating time domain (distribution of time between troubles). $f(t)dt$ is the probability of trouble in the interval dt. It is the time rate of change of probability of trouble.

Hazard rate

The instantaneous trouble rate $z(t)$. $z(t)$ is the time rate of change of probability of trouble, given that there has been no trouble in the time interval 0 to t. Thus, this conditional time rate of change of trouble is given by $z(t)=f(t)/R(t)$.

RELIABILITY PREDICTION AND QUALITY CONTROL

Background

Much of reliability theory is based on probability concepts which are independent of the physical form or characteristics of systems or devices. Since hardware has a long history relative to software, it is natural that almost all reliability literature and experience is based on the application of theory to hardware. Although certain modifications are necessary, it appears that important elements of reliability theory can validly be applied to software.

The classical model of reliability as applied to hardware involves three distinct periods in the life of equipment. During the burn-in period, when major bugs in equipment are identified and corrected or marginal components are forced to fail, equipment experiences a decreasing hazard rate. During this period the hazard rate is a function of the equipment operating time. In this period, the occurrence of failures *is dependent* on the failure history. If failures occur and are corrected prior to time t, this will have the effect of reducing the hazard rate at time t.

According to the classical model, following the burn-in period, failures are assumed to occur at a constant rate. This means that the occurrence of failures is *independent* of the age or operating time of the equipment. The expected number of failures during a given time interval is the same regardless of when the time interval starts, provided the equipment is operating within the constant hazard rate region. Failures within this region are said to occur suddenly or by chance, for example, when operating requirements or environmental requirements exceed the capabilities of the equipment. Another way of viewing the constant hazard rate region is that there is no preferred time of failure or time about which failures cluster.

The third region occurs when equipment is subject to rapid deterioration and wearout. This is the region of increasing hazard rate. During this period, the hazard rate is a function of operating time; hazard rate increases with time.

It has been found that certain probability density functions are appropriate for representing the distribution of time between failures or time to failure during the three regions of equipment life. For example, the two parameter (amplitude and shape) Weibull probability density function $f(t)=\alpha\beta t^{\beta-1}e^{-\alpha t^{\beta}}$ has a decreasing hazard rate function $z(t)=\alpha\beta t^{\beta-1}$, when $\beta<1$, where t is the time to failure or time between failures, α is an amplitude parameter and β is a shape parameter. The probability density and hazard rate functions can, in certain situations, be employed to represent the distribution of time between failures and the hazard rate, respectively, during the burn-in period.

During the operational, or constant hazard rate period, the time between failures is exponentially distributed. The exponential distribution corresponds to a Weibull distribution with $\beta=1$. Then, the probability density function and hazard rate are given by $f(t)=\alpha e^{-\alpha t}$ and $z=\alpha$, where α is the constant hazard rate and $1/\alpha$ is the mean time between failures.

Finally, during the wearout stage, when the hazard rate is increasing, a Weibull distribution with $\beta>1$ may be the appropriate distribution for representing time between failures. The log-normal and gamma (with appropriate choice of parameters) are other functions with an increasing hazard rate which may also be appropriate for this phase.

Application of Reliability Theory to Software

When applied to software reliability, many of the basic concepts and definitions of reliability theory remain intact. Among these are the following:

- Definition of reliability $R(t)$ as the *probability* of successful program operation for at least t hours
- Probability density function $f(t)$ of time between software troubles, or, equivalently, the time rate of change of the probability of trouble
- Hazard rate $z(t)$ as the instantaneous trouble rate, or, equivalently, the time rate of change of the conditional probability of trouble (time rate of change of probability of trouble, given that no trouble has occurred prior to time t)

There are also major differences between hardware and software reliability. These are listed below:

- Stresses and wear do not accumulate in software from one operating period to another as in the case of certain equipment; however, program quality may be different at the start of each run, for the reason given below.
- In the case of hardware, it is usually assumed that between the burn-in and wearout stages an exponential distribution (which means a constant hazard rate) applies and that the probability of failure in a time interval t is independent of equipment age. However, for software, there may be a difference in the initial "state of quality" between operating periods due to the correction of errors in a previous run or the introduction of new errors as the result of correcting other errors. Thus it is appropriate to employ a reliability growth model which would provide a reliability prediction at several points in the cumulative operating time domain of a program.
- For equipment, age is used as the variable for reliability prediction when the equipment has reached the wearout stage. Since with software, the concern is with running a program repeatedly for short operating times, the time variable which is used for reliability purposes is the time between troubles. However, cumulative operating time is a variable of importance for predicting the timing and magnitude of shifts in the reliability function as a result of the continuing elimination of bugs or program modification.

Over long periods of calendar or test time, there will

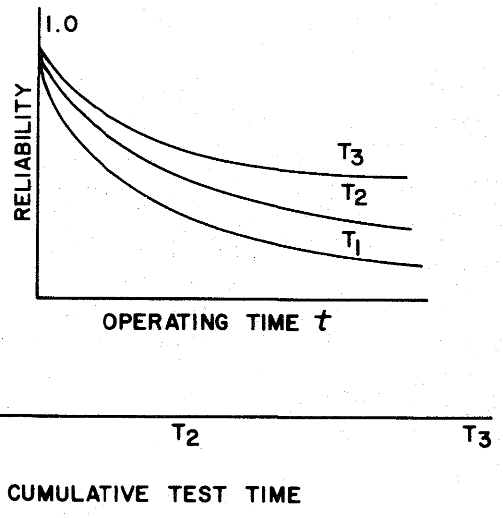


Figure 3—Reliability growth

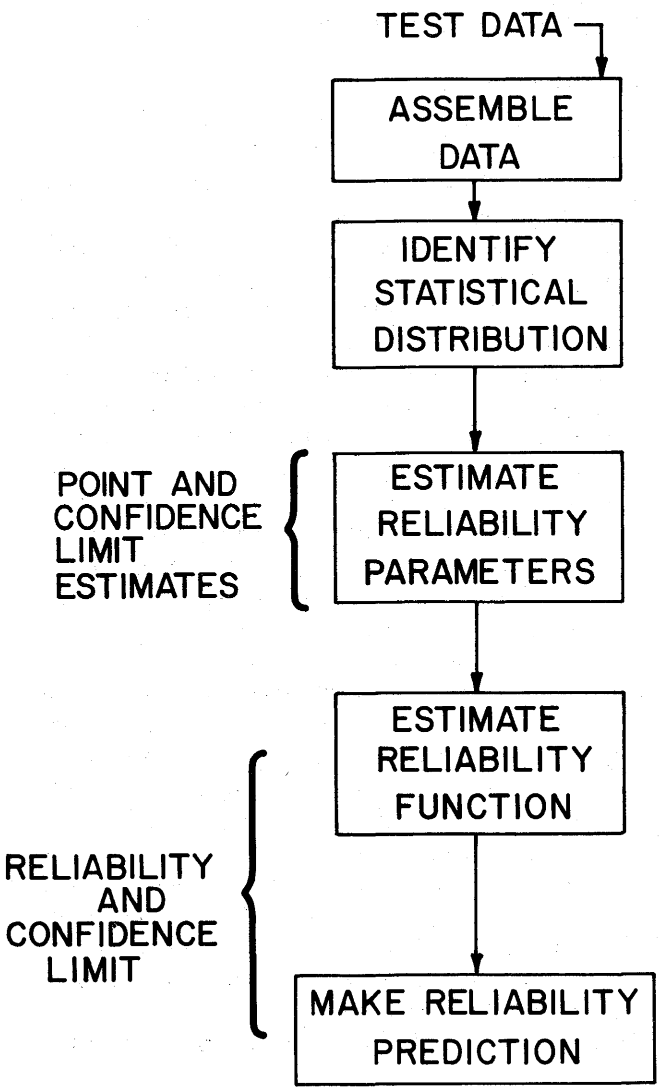

TEST DATA
ASSEMBLE DATA
IDENTIFY STATISTICAL DISTRIBUTION
POINT AND CONFIDENCE LIMIT ESTIMATES
ESTIMATE RELIABILITY PARAMETERS
ESTIMATE RELIABILITY FUNCTION
RELIABILITY AND CONFIDENCE LIMIT
MAKE RELIABILITY PREDICTION

Figure 4—Steps in reliability prediction

be shifts in the error occurrence process such that different hazard rate and probability density functions are applicable to different periods of time; or, the same hazard and probability density functions may apply but the parameter values of these functions have changed. This shift is depicted in Figure 3, where the reliability function, which is a decreasing function of *operating time* is shown shifted upward at various points in *cumulative test time,* reflecting long-term reductions in the trouble rate and an increase in the time between troubles.

Approach

The steps which are involved in one approach to software reliability prediction are shown in Figure 4.

Step 1. Assemble Data

Data must first be assembled in the form of a time between troubles distribution as was indicated in Figure 2. At this point, troubles are also classified by type and severity.

Step 2. Identify Statistical Distribution

In order to identify the type of reliability function which may be appropriate, both the empirical relative frequency function of time between troubles (see example in Figure 5) and the empirical hazard function are examined. The shapes of these functions provide qualitative clues as to the type of reliability function which may be appropriate. For example:

- A monotonically decreasing $f(t)$ and a constant $z(t)$ suggest an exponential function.
- An $f(t)$ which has a maximum at other than $t = 0$ and a $z(t)$ which monotonically increases suggests:
  —Normal function or
  —Gamma function or
  —Weibull function with $\beta > 1$.
- A monotonically decreasing $f(t)$ and a monotonically decreasing $z(t)$ suggest a Weibull function with $\beta < 1$.

After some idea is obtained concerning the possible distributions which may apply, point estimates of the parameters of these distributions are obtained from the sample data. This step is necessary in order to perform goodness of fit tests and to provide parameter

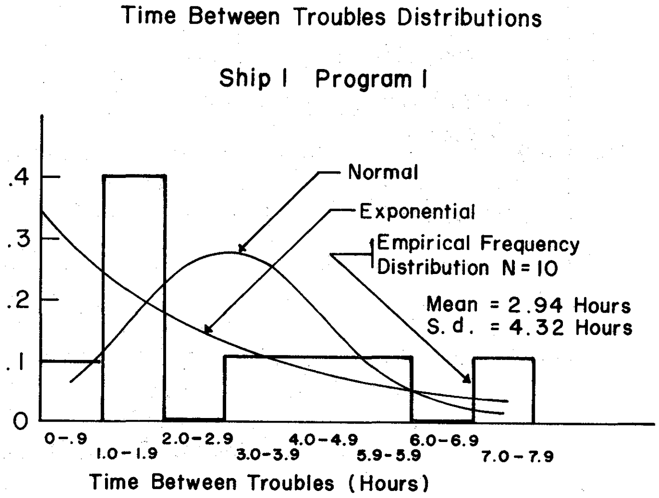


Figure 5—Probability density functions

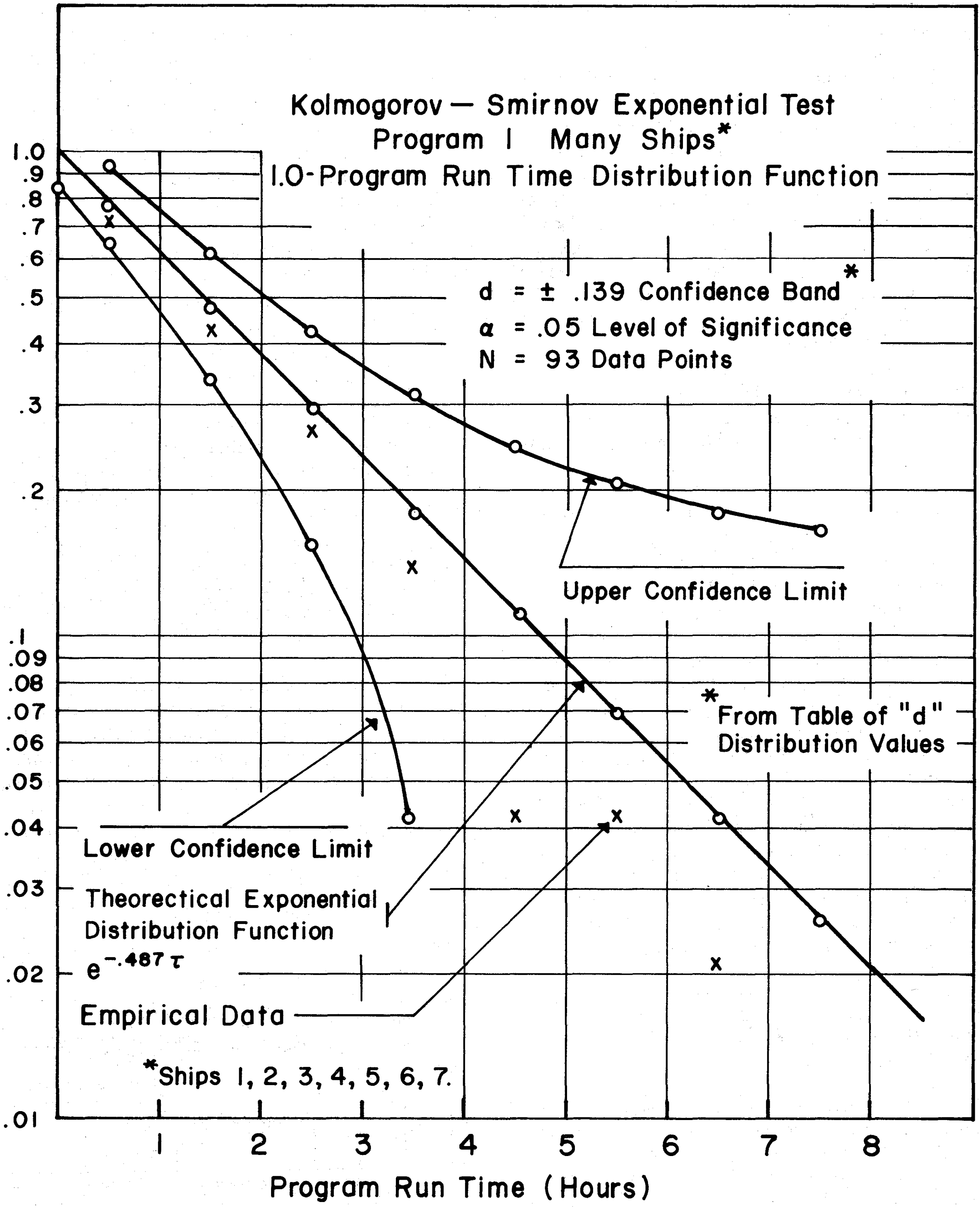


Figure 6—Goodness of fit test

estimates for the reliability function. In order to make a goodness of fit test, it is necessary to provide an estimate of the theoretical function to be used in the test. This is obtained by making point estimates of the applicable parameters. In the case of the one parameter exponential distribution, the point estimate would simply involve computing the mean time between troubles = total cumulative test time/number of troubles, which is the maximum likelihood estimator of the parameter T in the exponential probability density function $f(t) = 1/Te^{-t/I}$.

In the case of a multiple parameter distribution, the process is more involved. For the Weibull distribution, the following steps are required to obtain parameter point estimates:

- A logarithmic transformation of the hazard function is performed in order to obtain a linear function from which initial parameter values can be obtained.
- The initial values are used in the maximum likelihood estimating equations in order to obtain parameter point estimates.
- The probability density, reliability and hazard functions are computed using the estimated parameter values.

At this point, a goodness of fitness test can be performed between the theoretical probability density and the empirical relative frequency function or between the theoretical and empirical reliability functions. The Kolmogorov-Smirnov (K-S) or Chi Square tests can be employed for this purpose. An example of using the K-S test is shown graphically in Figure 6. This curve shows a test with respect to an exponential reliability function. Shown are the upper and lower confidence limits, the theoretical function and the empirical data points. Since the empirical points fall within the confidence band, it is concluded that the exponential is not an unreasonable function to employ in this case.

Step 3. Estimate Reliability Parameters Confidence Limits

The point estimate of a reliability parameter provides the best single estimate of the true population parameter value. However, since this estimate will, in general, differ from the population parameter value due to sampling and observational errors, it is appropriate to provide an interval estimate within which the population parameter will be presumed to lie. Since only the lower confidence limit of the reliability function is of interest, one-sided confidence limits of the parameters are computed. In Figure 7 is shown an example of the results of the foregoing procedure, wherein, for an exponential distribution, the point estimate of mean time between troubles (MTBT) is 2.94 hours (hazard rate of .34 troubles per hour) and the lower confidence limit estimate of MTBT is 2.27 hours (hazard rate of .44 troubles per hour). The lower confidence limit of MTBT for an exponential distribution is computed from the expression $T_i = 2n\bar{t}/\chi^2_{2n,1-\alpha}$ where T: is the lower confidence limit of MTBT, n is number of troubles, $\bar{t}$ is the MTBT (estimated from sample data), χ^2 is a Chi-Square value and α is the level of significance.

Step 4. Extimate Reliability Function

With point and confidence limit estimates of parameters available, the corresponding reliability functions can be estimated. The point and lower limit parameter estimates provide the estimated reliability functions $R = e^{-.34t}$ and $R = e^{-.44t}$, respectively, in Figure 7. In this example, the predicted reliabilities pertain to the occurrence of all categories of software trouble, i.e., the probability of no software troubles of any type occurring within the operating time of t hours.

Step 5. Make Reliability Prediction

With estimates of the reliability function available, the reliability for various operating time intervals can be predicted. The predicted reliability is then compared with the specified reliability. In Figure 7, the predicted reliability is less than the specified reliability (reliability objective) throughout the operating time range of the program. In this situation, testing must be continued until a point estimate of MTBT of 5.88 hours (.017 troubles per hour hazard rate) and a lower confidence limit estimate of MTBT of 4.55 hours (.022 troubles per hour hazard rate) are obtained. This result would shift the lower confidence limit of the predicted reliability function above the reliability objective.

Estimating test requirements

For the purpose of estimating test requirements in terms of test time and allowable number of troubles, curves such as those shown in Figure 8 are useful. This set of curves, applicable only to the exponential reliability function, would be used to obtain pairs of (test

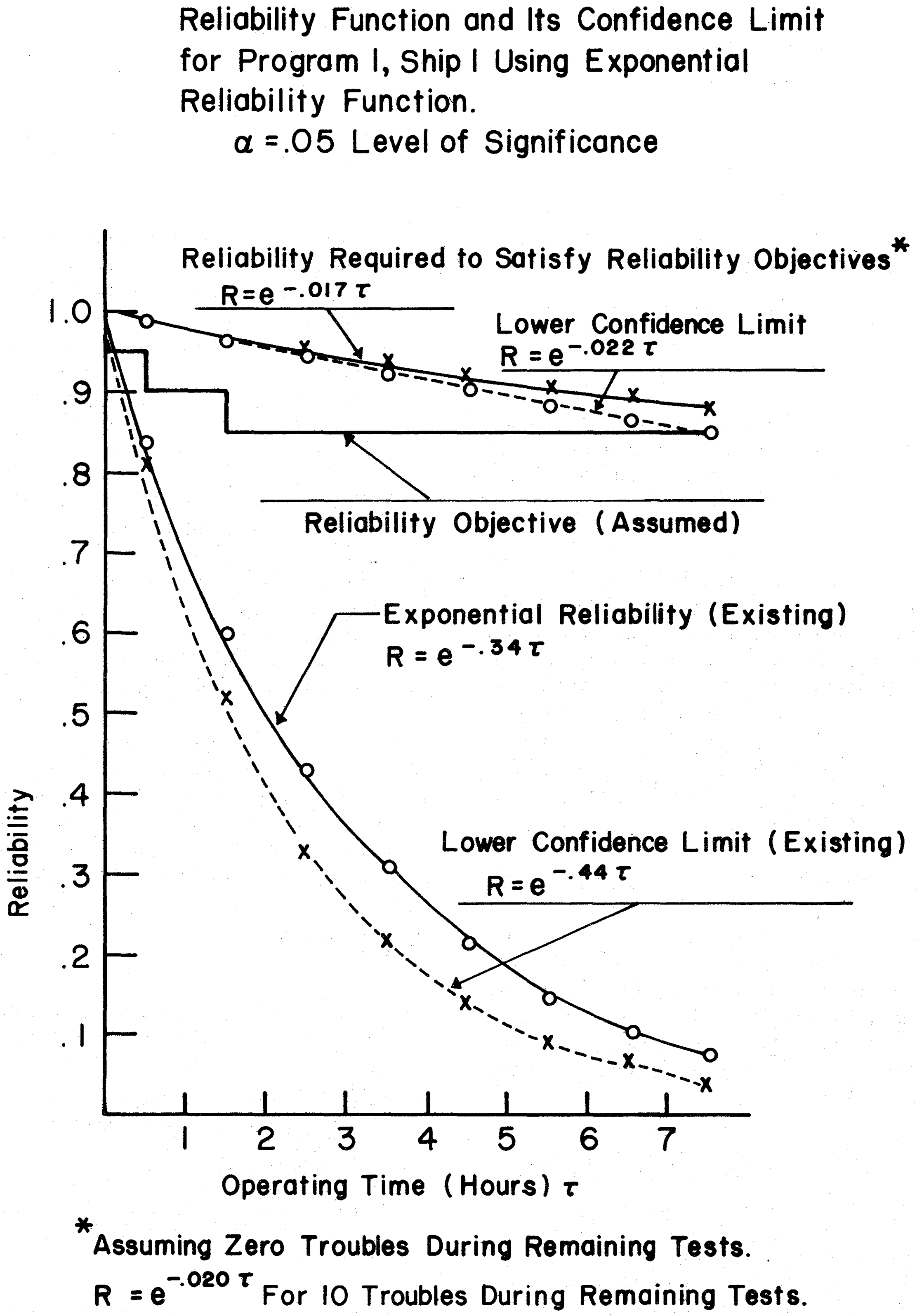


Figure 7—Reliability prediction

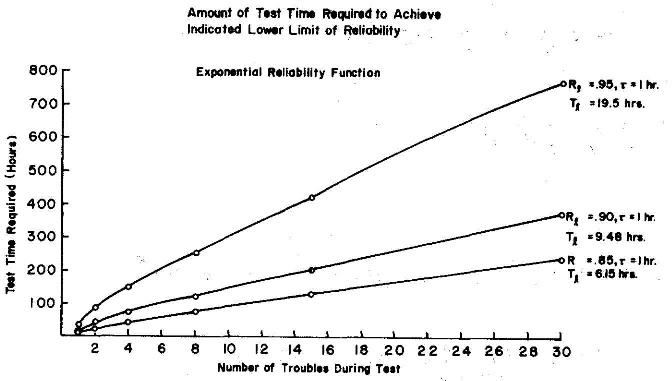


Figure 8—Test requirements

time, number of troubles) values. The satisfaction during testing of one pair of values is equivalent to satisfying the specified lower limit of reliability R_l for t hours of operation. For example, if a program reliability specification calls for a lower reliability confidence limit of .95 after 1 hour of *operating time*, this requirement would be satisfied by a *cumulative test time* of 100 hours and no more than 2 troubles; a cumulative test time of 200 hours and no more than 6 troubles; a cumulative test time of 300 hours and no more than 10 troubles, etc. The required test time is estimated from the relationship $T = [t\chi^2_{2n,1-\alpha}/2Ln(1/R_l)]$, where T is required test time, t is required operating time, χ^2 is a Chi Square value, n is number of troubles, R_l is the required lower limit of reliability and α is level of significance.

PRELIMINARY RESULTS AND CONCLUSIONS

A Naval Electronics Laboratory Center (NELC) sponsored study* was performed, employing the concepts and techniques described in this report, on Naval Tactical Data System (NTDS) data. The data utilized involved 19 programs, 12 ships and 325 software trouble reports. The major preliminary results and conclusions follow:

1. On the basis of Analysis of Variance tests, it was found that NTDS programs are heterogeneous with respect to reliability characteristics. There was greater variation of reliability between programs than within programs. This result suggests that program and programmer characteristics (source of between program variation) is more important in determining program reliability than is the stage of program checkout or cumulative test time utilized (source of within program variation). This result indicates a potential for obtaining a better understanding of the determinants of software reliability by statistically correlating program and programmer characteristics with measures of program reliability.
2. Goodness of fit tests indicated much variation among programs in the type of reliability function which would be applicable for predicting reliability. This result and the Analysis of Variance results suggest that program reliability should be predicted on an individual program basis and that it is not appropriate to merge sets of trouble report data from different programs in order to increase sample size for reliability prediction purposes.
3. Based on its application to NTDS data, the approach for reliability prediction and quality control which has been described appears feasible. However, the methodology must be validated against other test and operational data. Several interactive programs, written in the BASIC language, which utilize this approach, have been programmed at NELC*.

Another model by Jelenski and Moranda* has been developed and validated against NTDS and NASA data. Other approaches, such as reliability growth models, multiple correlation and regression studies and utilization of data smoothing techniques will be undertaken as part of a continuing research program.

BIBLIOGRAPHY

1 R M BALZER
EXDAMS—extendable debugging and monitoring system
AFIPS Conference Proceedings Vol 34 Spring 1969
pp 567–580

2 W J CODY
Performance testing of function subroutines
AFIPS Conference Proceedings Vol 34 Spring 1969
pp 759–763

3 J C DICKSON et al
Quantitative analysis of software reliability
Proceedings—Annual Reliability and Maintainability Symposium San Francisco California 25–27 January 1972
pp 148–157

* N. F. Schneidewind, "A Methodology for Software Reliability Prediction and Quality Control," *Naval Postgraduate School,* Report No. NPS55SS72032B, March 1972.

* Programmed by Mr. Craig Becker of the Naval Electronics Laboratory Center.

* Jelenski, Z. and Moranda, P. B., "Software Reliability Research," McDonnell Douglas Astronautics Company Paper WD1808, Navember 1971.

4 BERNARD ELSPOS et al
Software reliability
Computer January–February 1971 pp 21–27

5 ARNOLD F GOODMAN
The interface of computer science and statistics
Naval Research Logistics Quarterly Vol 18 No 2 1971 pp 215–229

6 K U HANFORD
Automatic generation of test cases
IBM Systems Journal Vol 9 No 4 1970 pp 242–256

7 Z JELINSKI P B MORANDA
Software reliability research
McDonnell Douglas Astronautics Company Paper WD 1808 November 1971

8 JAMES C KING
Proving programs to be correct
IEEE Transactions on Computers Vol C-20 No 11 November 1971 pp 1331–1336

9 HENRY C LUCAS
Performance evaluation and monitoring
Computing Surveys Vol 3 No 3 September 1971 pp 79–91

10 R B MULOCK
Software reliability engineering
Proceedings—Annual Reliability and Maintainability Symposium San Francisco California 25–27 January 1972 pp 586–593

11 R J RUBEY R F HARTWICK
Quantitative measurement of program quality
Proceedings—1968 ACM National Conference pp 672–677

12 N F SCHNEIDEWIND
A methodology for software reliability prediction and quality control
Naval Postgraduate School Report No NPS55SS72032B March 1972

The impact of problem statement languages on evaluating and improving software performance

by ALAN MERTEN and DANIEL TEICHROEW

The University of Michigan
Ann Arbor, Michigan

INTRODUCTION

The need to improve the methods by which large software systems are constructed is becoming widely recognized. For example, in a recent study (Office of Management and Budget[1]) to improve the effectiveness of systems analysts and programmers, a project team stated that: "The most important way to improve the effectiveness of the government systems analysts and programmers is by reducing the TIME now spent on systems analysis, design, implementations, and maintenance while maintaining or improving the present level of ADP system quality."

As another example, a study group (U.S. Air Force[2]) concluded that to achieve the full potential of command and control systems in the 1980's would require research and development in the following aspects of system building: requirements analysis and design, software system certification, software timeliness and flexibility.

Software evaluation is one part, but only a part of the total process of building information systems. The best way to make improvements is to examine the total process. This has been pointed out elsewhere (Teichroew and Sayani[3]) and by many others; Lehman,[4] for example, states:

"When first introduced, computers did not make a significant impact on the commercial world. The real breakthrough came only in the 1950's when institutions stopped asking, 'Where can we use computers?' and started asking, 'How shall we conduct our business now that computers are available?'

In seeking to automate the programming process, the same error has been committed. The approach has been to seek possible applications of computers within the process as now practiced.... Thus the problem of increasing programming effectiveness, through mechanization and tooling, is closely associated with the overall problem of methodology. Its solution calls for a review of the process itself, so that maximum benefit can be had from the use of computers."

This paper is concerned with one technique—problem statement languages—which are in accordance with the above philosophy as regards the system building system. They are an answer to the question: "How shall we conduct systems building now that computers are available?"

Problem statement languages are a class of languages designed to permit statement of requirements for information systems without stating the processing procedures that will eventually be used to achieve the desired results. Problem statement languages are used to formalize the definition of requirements, and problem statement analyzers can be used to aid in the verification of the requirement definition. These languages and analyzers have a potential for improving the total process of system building; this paper, however, will be limited to their role in software evaluation.

Software systems can be evaluated in three distinct ways. User organizations are primarily interested in whether the software produced performs the tasks for which it was intended. This first evaluation measure is referred to as the *validity* of the software. "Invalid" software is usually the result of inadequate communication between the user and the information system designers. Even in the presence of perfect communication, the software system often does not initially meet the specifications of the user. The second evaluation measure of software systems is their *correctness*. Software is correct if for a given input it produces the output implied by the user specifications. "Incorrect" software is usually the result of programming or coding errors.

A software system might be both valid and correct but still might be evaluated as being inefficient either by the user or the organization responsible for the computing facility. This characteristic is termed *performance* and is measured in terms of the amount of resources required by a software package to produce a result. The processing procedures and programs and files might be poorly designed such that the system costs too much to run and/or makes inefficient use of the computing resources. Poor performance of software is usually the result of inadequate attention to design or incorrect information on parameters that affect the amount of resources used.

Before discussing the value of problem statement languages in software evaluation, the concept is described briefly. The impact on software validity is also discussed. Once requirements in a problem statement language are available, it becomes possible to provide computer aids to the analysis and programming process which reduce the possibility of introducing errors. This impact of problem statement languages on software correctness is discussed later in the paper. In general, a given set of requirements can be implemented on a given complement of hardware in more than one way and each way may use a different amount of resources. The potential of problem statement languages to improve software performance is examined. Some preliminary conclusions from work to date on problem statement languages and analyzers related to software evaluation, and the impact of software evaluation on the design and use of problem statement languages and analyzers are discussed.

PROBLEM STATEMENT LANGUAGES AND PROBLEM STATEMENT ANALYZERS

Problem statement languages were developed to permit the problem definer (i.e., the analyst or the user) to express the requirements in a formal syntax. Examples of such languages are: a language developed by Young and Kent;[5] Information Algebra;[6] TAG;[7] ADS;[8] and PSL.[9] All of these languages are designed to allow the problem definer to document his needs at a level above that appropriate to the programmer; i.e., the problem definer can concentrate on *what* he wants without saying *how* these needs should be met.

A problem statement language is not a general-purpose programming language, nor, for that matter, is it a programming language. A programming language is one that can be used by a programmer to communicate with a machine through an assembler or a compiler. A problem statement language, on the other hand, is used to communicate the need of the user to the analyst. The problem statement language consequently must be designed to express what is of interest to the user; what outputs he wishes from the system, what data elements they contain, and what formulas define their values and what inputs are available. The user may describe the computational procedures and/or decision rules that must be used in determining the values of certain intermediate or output values. In addition, the user must be able to specify the parameters which determine the volume of inputs and outputs and the conditions (particularly those related to time) which govern the production of output and the acceptance of input.

These languages are designed to prevent the user from specifying processing procedures; for example, the user cannot use statements such as SORT (though he is allowed to indicate the order in which outputs appear), and he cannot refer to physical files. In some cases the languages are forms oriented. In these cases, the analyst using the problem statement language communicates the requirements by filling out specific columns of the forms used for problem definition. Other problem statement languages are free-form.

The difficulty of stating functional specifications in many organizational systems has been well recognized. (Vaughan[10]): "When a scientific problem is presented to an analyst, the mathematical statement of the relationships that exist between the data elements of the system is part and parcel of the problem statement. This statement of element relationships is frequently absent in the statement of a business problem. The seeming lack of mathematical rigor has led us to assume that the job of the business system designer is less complex than that of the scientific designer. Quite the contrary—the job of the business system designer is often rendered impossible because the heart of the problem statement is missing!

The fact that the relationships between some of the data elements of a business problem cannot be stated in conventional mathematical notation does not imply that the relationships are any less important or rigorous than the more familiar mathematical ones. These relationships form a cornerstone of any system analysis, and the development of a problem statement notation for business problems, similar to mathematical notation, could be of tremendous value."

Sometimes the lack of adequate functional specifications is taken fatalistically as a fact of life: An example is the following taken from a recent IBM report:[11]

"In practice, however, the functional specifications for a large project are seldom completely and consistently defined. Except for nicely posed mathematical requirements, the work of completely defining a functional specification usually approximates the work of coding the functions themselves. Therefore, such specifications are often defined within a specific problem context, or left for later detailed description. The detailed definition of functional specifications is usually unknown at the outset of the project. Much of the final detail is to be filled in by the coding process itself, based on a general idea of intentions. Hopefully, but not certainly, the programmers will satisfy these intentions with the code. Even so, a certain amount of rework can be expected through misunderstandings.

As a result of these logical deficiencies, a large programming project represents a significant management problem, with many of the typical symptoms of having to operate under conditions of incomplete and imperfect information. The main content of a programming project is logical, to be sure. But disaster awaits an approach that does not recognize illogical requirements that are bound to arise."

Problem statement languages can reduce the existence of illogical requirements due to poor specification. Despite the well-recognized need for formal methods of stating requirements and the fact that a problem statement language was published by Young and Kent in 1958, such languages are not in wide use today. One reason is that, until recently, there did not exist any efficient means to analyze a problem definition given in a problem statement language. Therefore, these languages were only used for the documentation of each user's requirements. Under these conditions, it was difficult to justify the expense of stating the requirements in this formal manner.

A problem statement language, therefore, is insufficient by itself. There must also be a formal procedure, preferably a software package, that will manipulate a problem statement for human use. Mathematics is a language for humans—humans can, after some study, learn to comprehend it and to use it. It is not at all clear that the equivalent requirements language for business will be manipulatable by humans, though obviously it must be understandable to them. Computer manipulation is necessary because the problem is so large, and a person can only look at one part at a time. The number of parameters is too large and their interrelationship is too complex.

A problem statement language must have sufficient structure to permit a problem statement to be analyzed by a computer program, i.e., a problem statement analyzer. The inputs and outputs of this program and the data base that it maintains are intended to serve as a central resource for all the various groups and individuals involved in the system building process.

Since usually more than one problem definer is required to develop requirements in an acceptable time frame, there must be provision for someone who oversees the problem definition process to be able to identify individual problem definitions and coordinate them; this is done by the problem definition management. One desirable feature of a system building process is to identify system-wide requirements so as to eliminate duplication of effort; this task is the responsibility of the system definer. Also, since the problem defines should use common data names there has to be some standardization of their names and characteristics and their definition (referred to here as "functions"). One duty of the data administrator is to control this standardization. If statements made by the problem definer are not in agreement as seen by the system definer or data administrator, he must receive feedback on his "errors" and be asked to correct these.

All of these capabilities must be incorporated in the problem statement analyzer which accepts inputs in the problem statement language and analyzes them for correct syntax and produces, among other reports, a comprehensive data dictionary and a function dictionary which are helpful to the problem definer and the data administrator. It performs static network analysis to ensure the completeness of the derived relationships, dynamic analysis to indicate the time-dependent relationships of the data, and an analysis of volume specifications. It provides the system definer with a structure of the problem statement as a whole. All these analyses are performed without regard to any computer implementation of the target information processing system. When these analyses indicate a complete and error-free statement of the problem, it is then available in two forms for use in the succeeding phases. One, the problem statement itself becomes a permanent, machine-readable documentation of the requirements of the target system *as seen by the problem definer* (not as seen by the programmer). The second form is a coded statement for use in the physical systems design process to produce the description of the target system as seen by the programmer.

A survey of problem statement languages is given in Teichroew.[12] A description of the Problem Statement Languages (PSL) being developed at The University of Michigan is given in Teichroew and Sibley.[9] In this paper the terms problem statement language and problem statement analyzer will refer to the general class while PSL and PSA will be used to mean the specific items being developed in the ISDOS Project at The University of Michigan.

ROLE OF PROBLEM STATEMENT LANGUAGES IN SOFTWARE VALIDITY

Definition of software validity

Often when software systems are completed, they do not satisfy the requirement for which they were intended. There may be several reasons for this. The user may claim that his requirements have not been satisfied. The systems analysts and programmers may claim that the requirements were never stated precisely or were changing throughout the development of the system. If the requirements were not precisely and correctly stated, the analysts and programmer may produce software which does not function correctly.

Software will be said to be valid if a correct, complete and precise statement of requirements was communicated to the analysts and programmers. Software which does not produce the correct result is said to be invalid if the reason is an error or incompleteness in the specification.

Problem statement languages can increase software validity by facilitating the elimination and detection of logical errors by the user, by permitting the use of the computer to detect logical errors of the clerical type and by using the computer to carry out more complex analysis than would be possible by manual methods.

Elimination or detection of logical errors by the user

Problem statement languages and analyzers appear to be one way to increase the communication between the user organizations and the analysts and programmers. Usually organizations find it very difficult to distinguish between their requirements and various procedures for accomplishing those requirements. This difficulty is often the result of the fact that the user has had some exposure to existing information processing procedures and attempts to use this knowledge of techniques to "aid" the analyst and in the development of the new or modified information system.

The major purpose of problem statement languages is to force the user to state only his requirements in a manner which does not force a particular processing procedure. Experience has shown that this requirement of problem statement languages is initially often difficult to impose upon the user. Often, he is accustomed to thinking of his data as stored in files and his processing requirements as defined in various programs written in standard programming languages. He has begun to think that his requirements are for programs and files and not for outputs of the system.

There is an interesting parallel between the use of problem statement languages and the use of data base management systems. Many organizations have found it difficult for users to no longer think in terms of physical stored data, but to concentrate on the logical structure of data and leave the physical storage to the data base management system. In the use of problem statement languages the user must think only of the logical data and the processing activities.

Initial attempts to encourage the use of problem statement languages have indicated some reluctance on the part of users to state only requirements, particularly if the "user" is accustomed to a programming language. However, once the functional analysts (problem definers) become familiar with the problem statement technique and learn to use the output from the problem statement analyzers, they find that they are able to concentrate on the specification of input and output requirements without having to be concerned about the design and implementation aspects of the physical systems. Similarly, output of problem statement analyzers can be used to aid the physical systems designers in the selection of better processing procedures and file organizations. The physical systems designer has the opportunity to look at the processing requirements and data requirements of all the users of the system and can select something that approaches global optimality as opposed to a design which is good for only one user.

One of the major problems in the design of software systems is the inability of users to segment the problem into small units to be attacked by different groups of individuals. Even when the problem can be segmented, the direct use of programming languages and/or data management facilities requires a great amount of interaction between the user groups and the designers throughout problem definition and design. Problem statement languages allow the individual users to state their requirements on a certain portion of the information system without having to be concerned with the requirement's definition of any other portion of the information systems.

This requirement is slightly modified in organizations where there exists a data directory (i.e., a listing of standard data names and their definition.) In this case each of the users must define his requirements in terms of standard names given in the data directory. In this case it is the purpose of the problem statement analyzer to check each problem statement to determine if it is using the previously approved data names. Besides the requirement to use standard data names, the individual problem definers can proceed with their problem definition in terms of their inputs, outputs, and processing procedures without knowledge of related data and processing activities. It is the purpose of the problem statement analyzer to determine the logical consistency of the various processing activities. It has been found that the individual problem definer might modify his statement of requirements upon receipt of the output of the problem statement analyzer. At this time he has an opportunity to see the relationship between his requirements and those of others.

Use of that problem statement analyzer to detect logical errors of the clerical type

Given that requirements are stated precisely in a problem statement language, it is possible to detect many logical errors during the definition stage of system development. Traditionally, these errors are not detected until much later, i.e., until actual programs and files have been built and the first stages of testing have been completed. Problem statement analyzers such as the analyzers built for ADS and for PSL at The University of Michigan can detect errors such as computation and storage of a data item that is not required by anyone, and the inputting of the same piece of data from multiple sources. Extensions of these analyzers will be able to detect more sophisticated errors. For example, they might detect a requirement for an output requiring a particular item prior to the time at which the item is input or can be computed from available input.

The problem analyzer can be used to check for problem definition errors by presenting information to a problem definer in a different order than it was initially collected. A data directory enumerates all the places in which a user-defined name appears. Another report brings together all the places where a system parameter has been used to specify a volume. An analyst can, by glancing through such lists, more readily detect a suspicious usage.

Complex analysis of requirements

The use of the computer as described above is for relatively routine clerical operations which could, in theory, be done manually if sufficient time and patience were available. The computer can also be used to carry out analysis of more complicated types or at least provide an implementation of heuristic rules. Examples are the ability to detect duplicate data names or to identify synonyms. These capabilities require an analysis of use and production of the various basic data items.

ROLE OF PROBLEM STATEMENT LANGUAGES IN SOFTWARE CORRECTNESS

Definition of software correctness

Software is said to be incorrect if it produced the wrong results even though the specification for which it was produced is valid and when the hardware is working correctly. The process of producing correct programs can be divided into five major parts:

—designing an algorithm that will accomplish the requirements
—translating the algorithm into source language
—testing the resulting program to determine whether it produces correct results
—debugging, i.e., locating the source of the errors
—making the necessary changes.

Current attempts to improve software correctness

Software incorrectness is a major cause of low programmer productivity. For example, Mills[11] states:

"There is a myth today that programming consists of a little strategic thinking at the top (program design), and a lot of coding at the bottom. But one small statistic is sufficient to explode that myth—the number of debugged instructions coded per man-day in a programming project ranges from 5 to at most 25. The coding time for these instructions cannot exceed more than a few minutes of an eight-hour day. Then what do programmers do with their remaining time? They mostly debug. Programmers usually spend more time debugging code than they do writing it. They are also apt to spend even more time reworking code (and then debugging that code) due to faulty logic or faulty communication with other programmers. In short, it is the thinking errors, even more than the coding errors which hold the productivity of programming to such low levels."

It is therefore not surprising that considerable effort has been expended to date to improve software correctness. Among the techniques being used or receiving attention are the following:

(1) Debugging aids. These include relatively simple packages such as cross-reference listers and snapshot and trace routines described in EDP Analyzer[13] and more extensive systems such as HELPER.[17]
(2) Testing aids. This category of aids includes module testers (e.g., TESTMASTER) and test data generators. For examples see EDP Analyzer[13] and the survey by Modern Data.[18]
(3) Structured and modular programming. This category consists of methodology standards and software packages that will hopefully result in

fewer programming errors in the first two phases—designing the algorithm and translating it to the source language code—and that they will be easier to find if they are made. (Armstrong,[19] Baker,[20] and Cheatham and Wegbreit.[21])

(4) Automated programming. This category includes methods for reducing the amount of programming that must be done manually by producing software packages that automatically produce source or object code. Examples are decisions table processors and file maintenance systems.[13,18]

Role of problem statement languages and analyzers in software correctness

While the need for the aids mentioned above has been recognized, there has been considerable resistance by programmers to their use.[13] Problem statement languages and problem statement analyzers can be of considerable help in getting programmer acceptance of such aids and in improving software correctness directly.

With the use of problem statement languages and analyzers, the programmer gets specifications in a more readable and understandable form and is, therefore, less likely to misinterpret them. In addition, extensions to existing analyzers could automatically produce source language statements. These extensions would take automatic programming methods to a natural limit since a problem statement is at least theoretically all the specification necessary to produce code. When the specifications are expressed in a problem statement language, the logical design of the system has effectively been decoupled from the physical design. Consequently, there is a much better opportunity to identify the physical processing modules. Once identified, they have to be programmed only once.

ROLE OF PROBLEM STATEMENT LANGUAGES IN SOFTWARE PERFORMANCE

Definition of software performance

Software which produces correct results in accordance with valid specification may still be rejected by the user(s) because it is too expensive or because it is not competitive with other software or with non-computerized methods. The "performance" of software, however, is a difficult concept to define.

One can give examples of performance measures in particular cases. For example, compilers are often evaluated in terms of the lines of source statements that can be compiled per minute on a particular machine. File organizations and access software (e.g., indexed sequential and direct access) are evaluated with respect to the rate at which data items can be retrieved. Sometimes software is compared on the basis of how computing time varies as a function of certain parameters. For example, matrix inversion routines are evaluated with respect to the relationship between process time and size of the array. Similarly, SORT packages are evaluated with respect to the time required to sort a given number of records.

Current methods to improve software performance

Software performance is important because any given piece of software is always in competition with the use of some other method, whether computerized or not. Considerable effort has been expended to develop methods to improve performance.

Software packages have been developed to improve the performance of programs stated in a general purpose language, either by separate programs, e.g., STAGED (OST) and CAPEX optimizer,[13] or incorporated directly into the compiler. Similar techniques are used in decision table processors.

A number of software packages developed can be used to aid in the improvement of performance of existing or proposed software systems. The software packages include software simulators and software monitors. Computer systems such as SCERT, SAM and CASE[14,15,16] can be used to measure the performance of a software/hardware system on a set of user programs and files. Another approach to improving performance of software systems is to measure the performance of the different components of an existing system either through a software monitor or by inserting statements in the program. The components which account for the largest amount of time can then be reworked to improve performance.

Each of these software aids attempts to improve the efficiency of a software system by modifying certain local code or specific file organizations. What would be more desirable is the ability to select the program and file organizations that best support the processing requirements of the entire information system.

Role of problem statement languages in software performance

Problem statement languages and analyzers can be used to improve the performance of software systems even beyond that feasible with the aids outlined above. Decisions involving the organization of files and the design of the processing procedures can be made based on the precise statement, and analysis, of the factors which influence the performance of the computing system because the problem statement requires explicit statement of time and volume information. The time information specifies the time at which input is received by the system or the time at which the specified output is required from the system. For a scheduled or periodic input, the time information specifies the time of the day, month, or year, or relative to some other calendar. For unscheduled or random input, the expected rate for a fixed time interval is specified. The volume information consists of specification of the "size" of the input or the output. For example, the number of time cards or the number of paychecks. Volume information is specified so that it is possible to determine the number of characters of data that will be stored or processed and moved between storage devices.

In order to design an efficient information system, the analyst must consider the processing needs of the individual users in arriving at a structure for the data base. Each of the data elements of the data base must be considered, and an indication made of the processing activities to be supported by the specific data. From this, the system designers must determine the file organization, i.e., the physical mapping of the data onto secondary storage. Methods for maintaining that data must be determined through the use of the time and volume information specified in the information system requirements.

Problem statement analyzers summarize, format and display the time and volume information which is relevant to the file designers. Our experience with problem statement languages seems to indicate that efficient systems are designed in which a file organization is initially determined and then the program design is undertaken. Problem statement languages are used to state the processing procedures required to produce the different output products or to process the various input data. Problem statement analyzers must have the ability to aid the systems analyst to group various data and also to group procedures in such a way as to minimize the amount of transfer of data between primary memory and secondary memory. One of the outputs from the problem statement analyzer such as the one for PSL is an analysis of the processing procedures which access the same or overlapping sets of data. These processes can, in many cases, be grouped together to form programs.

Currently software systems are often defined in which a file organization is initially selected based on some subset of the processing requirements. Following this selection, additional processing requirements are designed to "fit into" this initial file organization. As problem statement analyzers become more powerful, it will be possible to delay this decision concerning selection of a file organization until all the major processing requirements have been specified.

REMARKS

To our knowledge there does not exist any definitive study with a controlled experiment and collection of data to answer questions such as "why does software not produce the desired result; why does it not produce the correct result; and why does it not use resources efficiently?" However, it is generally agreed that the major causes include the following:

1. Errors in communication or misunderstandings from those who determine whether final results are valid, correct and produced efficiently to those who design and build the system.
2. Difficulties in defining interfaces in the programming process. The serious errors are not in individual programs, but in ensuring that the output of one program is consistent with the input to another.
3. Inability to test for all conceivable errors.

Considerable effort has gone into developing methods and packages to improve software—some of these have been mentioned in this paper. The ISDOS Project at The University of Michigan has been engaged in developing and testing several problem statement languages and problem statement analyzers. This paper has been concerned with the ways in which this use of problem statement languages and problem statement analyzers will lead to better software. Problem statement analyzers have been developed for ADS and for PSL at The University of Michigan. The analyzers have been used in conjunction with the development of certain operational software systems as well as a teaching and research tool in the University.

The ADS analyzer has been tested on both large and small problems to determine its use in software evaluation and development. The analyzer is currently being modified and extended to be used extensively by a government organization. Initial research concerning the installation of this analyzer within the organization indicates that analyzers must be slightly modified to interface with the procedures of the functional user.

Other portions of the analyzer appear to be able to be used as they currently exist. Generally, it appears as if analyzers will have a certain set of organizationally independent features and will have to be extended to include a specific set of organizationally dependent features for each organization in which they are used.

CONCLUSION

Many of the techniques being used or proposed to improve software performance are based on the current methods of developing software. The problem statement language represents an attempt to change the method of software development significantly by specifying software requirements. This paper has attempted to demonstrate that the use of problem statement languages and analyzers could improve software in terms of validity, correctness and performance.

The design of problem statement languages and the design and construction of problem statement analyzers are formidable research and development tasks. In some sense the design task is similar to the design of standard programming languages and the design and construction of compilers and other language processors. However, the task appears more formidable when one considers that these languages will be used by non-computer personnel and are producing output which must be analyzed by these people.

The procedure by which these techniques are tested and refined will probably be similar to the development and acceptance of the "experimental" compilers and operating systems of the 1950's and 1960's. These techniques are directed at the specification of requirements for application software. This phase of the life cycle of information systems has received the least amount of attention to date. The development and use of problem statement languages and analyzers can aid this phase. As the languages are improved and extended, their value as an aid to the entire process of software development should be realized.

ACKNOWLEDGMENTS

This research was supported by U.S. Army contract #010589 and U.S. Navy Contract #N00123-70-C-2055.

REFERENCES

1 OFFICE OF MANAGEMENT AND BUDGET
Project to improve the ADP systems analysis and computer programming capability of the federal government
December 17 1971 69 pp and Appendices

2 U.S. AIR FORCE
Information processing data automation implications of air force command and control requirements in the 1980's
Executive Summary February 1972 (SAMSO/XRS-71-1)

3 D TEICHROEW H SAYANI
Automation of system building
DATAMATION August 15 1972

4 M M LEHMAN L A BELADY
Programming systems dynamics or the metadynamics of systems in maintenance and growth
IBM Report RC 3546 September 17 1971

5 J W YOUNG H KENT
Abstract formulation of data processing problems
Journal of Industrial Engineering November-December 1958. See Also Ideas for Management International Systems-Procedures Assoc

6 CODASYL DEVELOPMENT COMMITTEE
An information algebra phase I report
Communications ACM 5 4 April 1962

7 IBM
The time automated grid system (TAG): sales and systems guide
Reprinted in J F Kelly Computerized Management Information System Macmillan New York 1970

8 H J LYNCH
ADS: A technique in system documentation
ACM Special Interest Group in Business Data Processing Vol 1 No 1

9 D TEICHROEW E SIBLEY
PSL, a problem statement language for information processing systems design
The University of Michigan ISDOS Working Paper June 1972

10 P H VAUGHAN
Can COBOL cope
DATAMATION September 1 1970

11 H D MILLS
Chief programmer teams: principles and procedures
IBM Federal Systems Division FSC 71-5108

12 D TEICHROEW
Problem statement languages in MIS
E Grochla (ed) Management-Information-Systeme Vol 14
Schriftenreiche Betriebswirteschaftliche Beitrage zur
Organisation und Automation Betriebswirtschaftlicher
Verlag Weisbaden 1971

13 EDP ANALYZER
COBOL aid packages
Canning Publications Vol 10 No 8 May 1972

14 J W SUTHERLAND
The configuration: today and tomorrow
Computer Decisions N J Hayden Publishing Company Rochelle Park February 1971

15 J W SUTHERLAND
Tackle system selection systematically
Computer Decisions N J Hayden Publishing Company Rochelle Park February 1971

16 J N BAIRSTOW
A review of systems evaluation packages
Computer Decisions N H Hayden Publishing Company Rochelle Park June 1970

17 R R RUSTIN (ed)

Debugging techniques in large systems
Prentice-Hall 1971

18 SOFTWARE FORUM
Survey of program package-programming aids
Modern Data March 1970

19 R ARMSTRONG
Modular programming for business applications
To be published

20 F T BAKER
Chief programmer team management of production programming
IBM Systems Journal No 1 1972

21 T E CHEATHAM JR B WEGBREIT
A laboratory for the study of automated programming
SJCC 1972

The solution of the minimum cost flow and maximum flow network problems using associative processing

by VINCENT A. ORLANDO and P. BRUCE BERRA*

Syracuse University
Syracuse, New York

INTRODUCTION

The minimum cost flow problem exists in many areas of industry. The problem is defined as: given a network composed of nodes and directed arcs with the arcs having an upper capacity, lower capacity, and a cost per unit of commodity transferred, find the maximum flow at minimum cost between two specified nodes while satisfying all relevant capacity constraints. The classical maximum flow problem is a special case of the general minimum cost flow problem in which all arc costs are identical and the lower capacities of all arcs are zero. The objective in this problem is also to find the maximum flow between two specific nodes. Algorithms exist for the solution of these problems and are coded for running on sequential computers. However, many parts of both of these problems exhibit characteristics that indicate it would be worthwhile to consider their solution by associative processors.

As used in this paper, an associative processor has the minimum level capabilities of content addressability and parallel arithmetic. The content addressability property implies that all memory words are searched in parallel and that retrieval is performed by content. The parallel arithmetic property implies that arithmetic operations are performed on all memory words simultaneously.

In this paper, some background in associative memories/processors and network flows, is first provided. We then present our methodology for comparison of sequential and associative algorithms through the performance measures of storage requirements and memory accesses. Finally we compare minimum cost flow and maximum flow problems as they would be solved on sequential computers and associative processors; and present our results.

ASSOCIATIVE MEMORIES/PROCESSORS[1-5]

The power of the associative memory lies in the highly parallel manner in which it operates. Data are stored in fixed length words as in conventional sequential processors, but are retrieved by content rather than by hardware storage address. Content addressing can take place by field within the storage word so, in effect, each word represents an n-tuple or cluster of data and the fields within each word are the elements, as illustrated in Figure 1. One of the ways in which accessing can take place is in a word parallel, bit serial manner in which all words in memory are read and simultaneously compared to the search criteria. This allows the possibility of retrieving all words in which a specified field satisfies a specified search criterion. These search criteria include equality, inequality, maximum, minimum, greater than, greater than or equal to, less than, less than or equal to, between limits, next higher and next lower. Further, complex queries can be formed by logically combining the above criteria. Boolean connectives include AND, inclusive OR, exclusive OR and complement. Finally, any number of fields within the word can be defined with no conceptual or practical increase in complexity. That is, within the limitation of the word length, any number of elements may be defined within a cluster.

In addition to the capabilities already mentioned, associative memories can be constructed to have the ability of performing arithmetic operations simultaneously on a multiplicity of stored data words. Devices of this type are generally called associative processors. Basic operations that can be performed on all words that satisfy some specified search criteria as previously described include: add, subtract, multiply or divide a constant relative to a given field; and add, subtract, multiply or divide two fields and place the result in a third field. This additional capability extends the use of the associative processor to a large

* This research partially supported by RADC contract AF 30602-70-C-0190 Large Scale Information Systems.

class of scientific problems in which similar operations are repeated for a multiplicity of operands.

While various architectures exist for these devices and they are often referred to by other names (parallel processors, associative array processors), in this paper we have adopted the term associative processor and further assume that the minimum level capabilities of the device include content addressing, parallel searching and parallel arithmetic as described above.

NETWORK DATA STRUCTURE

An example network is given in Figure 2, in which the typical arc shown has associated with it a number of attributes. The type and number of these attributes depends upon the specific network problem being solved but typically include the start node, end node, length, capacity and cost per unit of commodity transferred. In solving problems, considerably more than just the network definition attributes are required due to additional arc related elements needed in the execution of the network algorithms. Included are items such as node tags, dual variables, flow, bookkeeping bits, etc. Thus, each arc represents an associative cluster of data and hence can be stored within the associative memory with a minimum of compatibility problems.

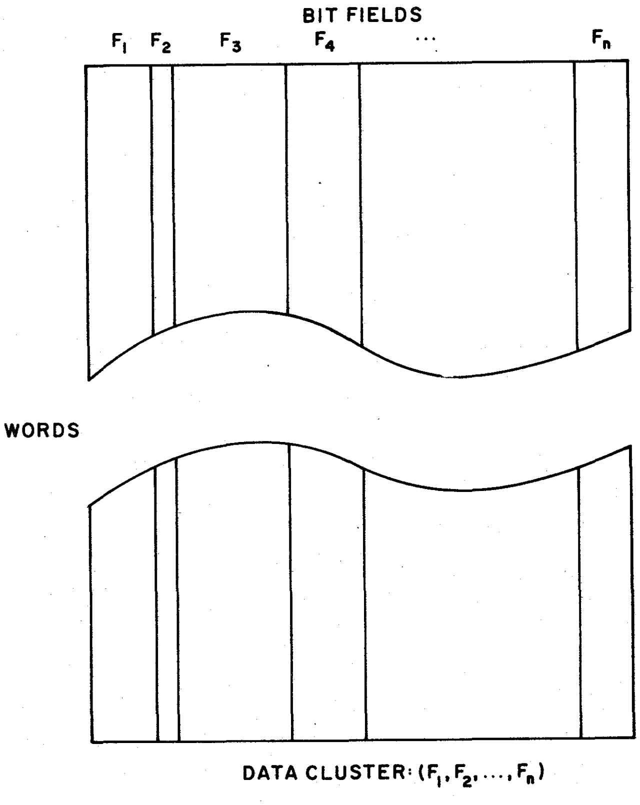


Figure 1—Associative memory layout

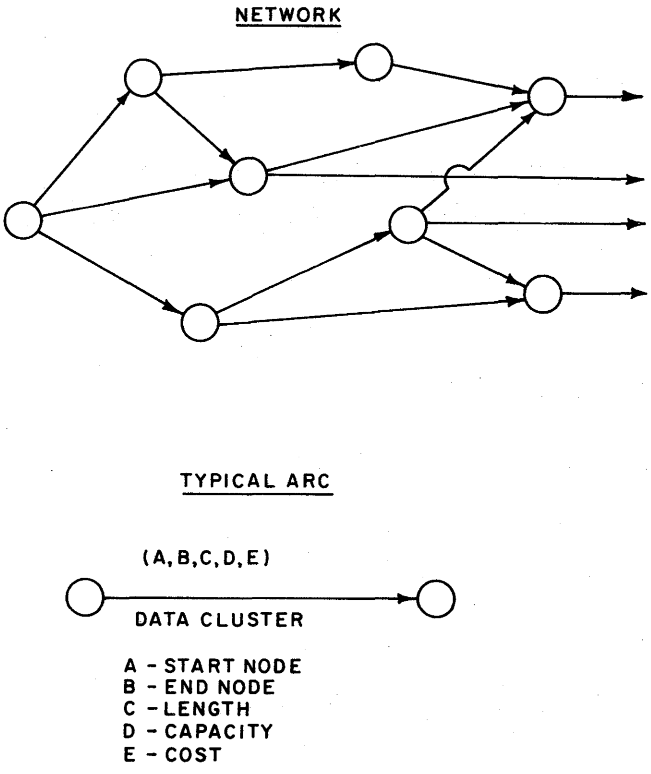


Figure 2—Data structure for network problems

The above discussion applies to network problems in general and served as the basis for research by the authors[6,7] into the use of associative processing in the solution of the minimum path, assignment, transportation, minimum cost flow and maximum flow problems. Results for all problems indicated that a significant improvement in execution time can be achieved through the use of the associative processor. The purpose of this paper is to describe the details of the methodology used and present the results obtained for the minimum cost flow and maximum flow problems.

METHODOLOGY AND MEASURES OF PERFORMANCE

The general methodology followed in this research was to first solve small problems on sequential computers in order to develop mathematical relationships that could be used to extrapolate to large problems; then to solve small problems on an associative processor emulator to again generate data that could be used in extrapolating to large problems and finally, to compare the results. This methodology had the distinct advantage of obtaining meaningful data without having to expend vast amounts of computer time in solving large problems. Further, since we did not have a hardware associative processor at our disposal, through the use of the emulator, we were able to solve real problems in the same way as with the hardware.

In order to compare the compiler level sequential program to the emulated associative program, it was first necessary to define some meaningful measures of performance. It was considered desirable for these measures to be implementation independent and yet yield information on relative storage requirements and execution times since these are the characteristics most often considered in program evaluation. Measures satisfying the above requirements which were used in the performance comparisons of this research are storage requirements and accesses.

The storage requirements measure is defined as the number of storage words required to contain the network problem data and any intermediate results. It should be noted that the number of bits per storage word would typically be greater for an associative processor since word lengths of 200 to 300 bits are typical of the hardware in existence or under consideration. However, word comparisons have the advantage of being implementation independent while providing a measure that is readily converted to the bit level for specific comparisons in which the word lengths of each machine are known. A determination of storage requirements for the competing programs was accomplished by counting the size of the arrays for the sequential programs and the number of emulator storage words for the associative programs. In both cases we assumed that enough memory was available to hold the entire problem.

The storage accesses measure is defined as the number of times that the problem data are accessed during algorithm execution. Defined in this manner this quantity is also implementation independent. However, it should be noted that the ratio of sequential to associative processor accesses is approximately equal to the ratio of execution times that would be expected if both algorithms were implemented on current examples of their respective hardware processors. This is true since the longer cycle time of the associative processor is more than offset by the large number of machine instructions represented by each of the sequential accesses. Collection of data for this measure was accomplished in the sequential programs by the addition of statements to count the number of times that the array problem data were accessed. Only accesses to original copies of the array data were included in this count. That is, accessing of non-array constants that temporarily contained problem data was not counted. Data collection for the associative programs was accomplished by a counter that incremented each time the emulator was called.

SEQUENTIAL ALGORITHM ANALYSIS

It was recognized that it would be highly desirable to obtain the sequential algorithms from an impartial, authoritative source, since this would tend to eliminate the danger of inadvertently using poor algorithms and thus obtaining results biased in favor of the associative processor. A search of the literature indicated that these requirements were perhaps best met by algorithms published in the *Collected Algorithms from the Association for Computing Machinery* (CACM)[8].

While these algorithms may not be the "best" in certain senses, they have the desirable property of being readily available to members of the computer field.

Algorithm 336[9] is the only algorithm published in the CACM that solves the general minimum cost flow problem stated above. This algorithm is based on the Fulkerson out-of-kilter method[10] which is the only approach available for a single phase solution to this problem. That is, this method permits conversion of an initial solution to a feasible solution (or indicates if none exists) at the same time that optimization is taking place. Other algorithms accomplish these tasks in separate and distinct phases.

The single algorithm published by the CACM for the maximum flow problem is number 324[11] which is based on the Ford and Fulkerson method.[12] This method appears to be recognized as a reasonable approach since it is consistently chosen for this problem in textbooks on operations research.[13,14]

These sequential algorithms were implemented in FORTRAN IV and executed on the Syracuse University Computing Center IBM 360/50 to verify correctness of the implementations and to collect performance data.

A detailed analysis of the logic for Algorithm 336 indicates that the access expressions for this program are as follows

$$\mathrm{NAC_{SEQ}} = 11\ \mathrm{NARCS} + \sum_{i=1}^{\mathrm{NB}} \mathrm{NAI_{SEQ}(BR)}_i + \sum_{j=1}^{\mathrm{NN}} \mathrm{NAI_{SEQ}(NON)}_j \quad (1)$$

where

$$\mathrm{NAI_{SEQ}(BR)}_i = \mathrm{N} + 21\ \mathrm{NLAB}_i + 4\ \mathrm{NONL1}_i + 13\ \mathrm{NONL2}_i + 9\ \mathrm{NAUG}_i - 30 \quad (2)$$

$$\mathrm{NAI_{SEQ}(NON)}_j = 4\ \mathrm{N} + 19\ \mathrm{NLAB}_j + 4\ \mathrm{NONL1}_j + 13\ \mathrm{NONL2}_j + 4\ \mathrm{NARCS} + 9\ \mathrm{NPLAB}_j - 12 \quad (3)$$

and

NAC = number of storage accesses required for problem solution

NAI(BR) = number of accesses in a flow augmenting iteration, called a breakthrough condition

NAI(NON) = number of accesses in an improved dual solution iteration, called a non-breakthrough condition

NB = number of breakthrough iterations in a problem

NN = number of non-breakthrough iterations in a problem

N = number of network nodes

NARCS = number of network arcs

NLAB = number of nodes labeled during an iteration

NONL1 = number of arcs examined during an iteration that have both nodes unlabeled

NONL2 = number of other arcs examined in an iteration that do not result in a labeled node

NPLAB = number of arcs with exactly one node labeled

NAUG = number of nodes on a flow augmenting path.

Note that the above expressions represent a non-typical best case for the sequential labeling process since it is assumed that only one pass through the list of nodes is required for the entire labeling process.

To simplify the above expressions, assume that all arcs processed which do not result in a labeled node are of type NONL1. This then makes NONL1 = NARCS − NLAB. Further assume that NPLAB takes on its average lower bound of NARCS/N. Both of these assumptions introduce further bias in favor of the sequential program. After making these substitutions, equations (2) and (3) become

$$\text{NAI}_{\text{SEQ}}(\text{BR})_i = \text{N} + 17\ \text{NLAB}_i + 4\ \text{NARCS} + 9\ \text{NAUG}_i - 30 \quad (4)$$

$$\text{NAI}_{\text{SEQ}}(\text{NON})_j = 4\ \text{N} + 15\ \text{NLAB}_j + 8\ \text{NARCS} + 9\ \text{NARCS/N} - 12 \quad (5)$$

In a similar manner, a best case access expression was developed for Algorithm 324 and is given as follows

$$\text{NAI}_{\text{SEQ}} := 3\ \text{N} + 8(\text{NARCS/N})(\text{NAUG}_i - 1) + 10\ \text{NAUG}_i + 4\ \text{NLAB}_i - 16 \quad (6)$$

where NAI = the number of accesses in an iteration.

The above access expressions were verified by comparing the predicted values with those obtained experimentally through actual execution of the programs.

ASSOCIATE ALGORITHM ANALYSIS

The out-of-kilter method described above was also used as the basis for the associative processor algorithm since it represents the only minimum cost flow method available that is developed from a network rather than a matrix orientation. The node tags which are used to define the unsaturated path from source to sink are patterned after the labeling method of Edmonds and Karp as described in Hu.[13] This selection was made to exploit the associative processor minimum search capability by finding the minimum excess capacity after the sink was reached, rather than performing a running comparison at each labeled node as in the original labeling method. For a discussion of the details of this development see Orlando.[6]

As indicated earlier, hardware implementation of the developed algorithm was not possible since very few associative processors are in existence and in particular none was available for this research. To circumvent this problem, as previously stated, a software interpretive associative processor emulator was developed after extensive investigation of the programming instruction format and search capabilities available on the Rome Air Development Center (RADC) Associative Memory.[15]

Additional arithmetic capabilities expected to be available on any associative processor were included in the emulator. Thus, it had the basic properties of content addressability, parallel search and parallel arithmetic.

In operation, the associative network programs, composed primarily of calls to the emulator, are decoded and executed one line at a time. Each execution, although composed of many sequential operations, performs the function of one associative processor statement written at the assembly language level. The program for the emulator was implemented in FORTRAN IV and executed on the IBM 360/50. Complete details of the capabilities and operation of the emulator and listings of the associative emulator programs are contained in Orlando.[6]

The access expressions for the associative processor program derived through a step by step analysis of the logic are presented below. The terminology used is the same as defined previously.

$$\mathrm{NAC_{AP}} = 3 + \sum_{i=1}^{\mathrm{NB}} \mathrm{NAI_{AP}(BR)}_i + \sum_{j=1}^{\mathrm{NN}} \mathrm{NAI_{AP}(NON)}_j \qquad (7)$$

where

$$\mathrm{NAI_{AP}(BR)}_i = 13\ \mathrm{NLAB}_i + 3\ \mathrm{NAUG}_i + 20 \qquad (8)$$

$$\mathrm{NAI_{AP}(NON)}_j = 13\ \mathrm{NLAB}_j + 29. \qquad (9)$$

The above access expressions represent a worst case for the algorithm logic since each step includes the maximum amount of processing possible. That is, it was assumed that the out-of-kilter arc detected always belonged to the last case to be tested and that each node used as a base point for labeling only resulted in the labeling of one additional node.

The associative processor algorithm for the maximum flow problem is based on the Ford & Fulkerson method[12] with the modification of node labeling as described above. A comparable worst case access expression for this algorithm is

$$\mathrm{NAI_{AP_i}} = 11\ \mathrm{NLAB}_i + 3\ \mathrm{NAUG}_i \qquad (10)$$

The above access expressions were also verified using experimental data obtained from execution of the emulated programs.

PERFORMANCE COMPARISON

The list orientation of the sequential program for the minimum cost flow problem imposes a requirement of 7 NARCS+N words for the storage of problem data. This is approximately seven times the NARCS+1 storage words required by the associative processor program. However, since both programs store network data in the form of arc information, the above comparison is the same for all networks.

Access comparisons between the sequential and associative processor programs are made on an average per iteration basis. This eliminates the need to assume values for the number of breakthrough and non-breakthrough iterations needed for problem solution. This approach is valid in terms of total problem access requirements since both algorithms are based on the same method and would therefore require the same number of each type of iteration in the solution of the same problem. The main effect of this approach is to eliminate from the comparison the number of accesses required for problem initialization. From equation (1) it is seen that 11 NARCS accesses are required by the sequential program for this purpose while equation (7) shows that the associative processor program requires 3 accesses for problem initialization regardless of network size. Thus, the comparison on an iteration basis introduces an additional bias in favor of the sequential program.

In order to avoid handling the breakthrough and non-breakthrough cases separately, the comparison will be made on the basis of an average of breakthrough and non-breakthrough access requirements. That is, change to mean values and define

$$\overline{\mathrm{NAI}} = \frac{\overline{\mathrm{NAI}}\mathrm{(BR)} + \overline{\mathrm{NAI}}\mathrm{(NON)}}{2}. \qquad (11)$$

Experience with the algorithm indicates that in general the majority of the problem iterations result in non-breakthrough and therefore the average as defined in equation (11) gives this case a smaller than realistic weighting. A comparison of the iteration access expressions, equations (4), (5), (8) and (9) indicate a greater relative performance gain for the associative processor in the breakthrough case. Therefore, the equal weighting of the iteration types introduces additional bias in favor of the sequential program.

Substitution of the access expressions in equation (11) yields

$$\overline{\mathrm{NAI}}_{\mathrm{SEQ}} = \tfrac{1}{2}(5\ \mathrm{N} + 32\ \overline{\mathrm{NLAB}} + 12\ \mathrm{NARCS} + 9\ \overline{\mathrm{NAUG}} + 9\ \mathrm{NARCS/N} - 42) \qquad (12)$$

$$\overline{\mathrm{NAI}}_{\mathrm{AP}} = \tfrac{1}{2}(26\ \overline{\mathrm{NLAB}} + 3\ \overline{\mathrm{NAUG}} + 49). \qquad (13)$$

Now, let $\overline{\mathrm{NLAB}} = \mathrm{aN}$ and $\overline{\mathrm{NAUG}} = \mathrm{bN}$ which by

TABLE I—Minimum Cost Flow Access Performance Data

		ASSO-CIA-TIVE	SE-QUEN-TIAL		
N	NARCS	NAI	NAI	R	D
100	100	1,475	2,864	2.0	.01
	1,000		8,304	5.6	.10
	6,000		38,529	26.1	.61
	10,000		62,706	42.5	1.00
500	500	7,275	14,464	2.0	.002
	1,000		17,468	2.4	.004
	10,000		71,549	9.8	.04
	100,000		612,359	84.2	.40
	150,000		912,809	125.5	.60
	250,000		1,513,709	208.1	1.00
1,000	1,000	14,525	28,964	2.0	.001
	10,000		83,004	5.7	.01
	100,000		623,409	42.9	.10
	600,000		3,625,659	249.7	.60
	1,000,000		6,027,459	415.0	1.00

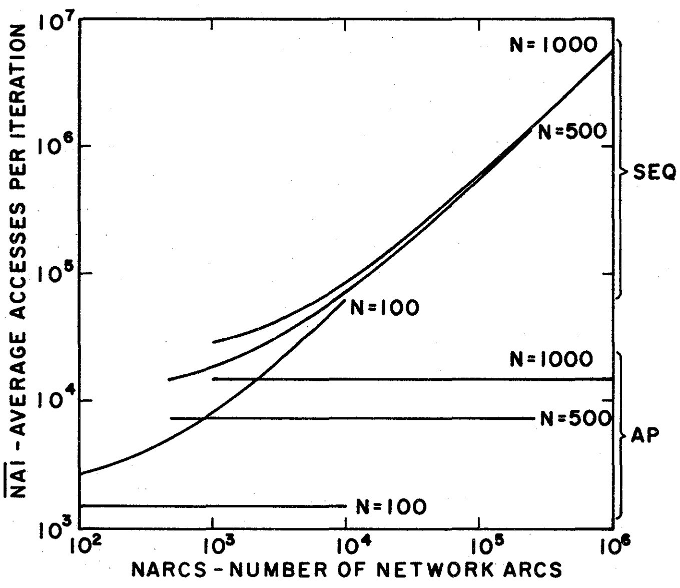


Figure 3—Minimum cost flow access requirements

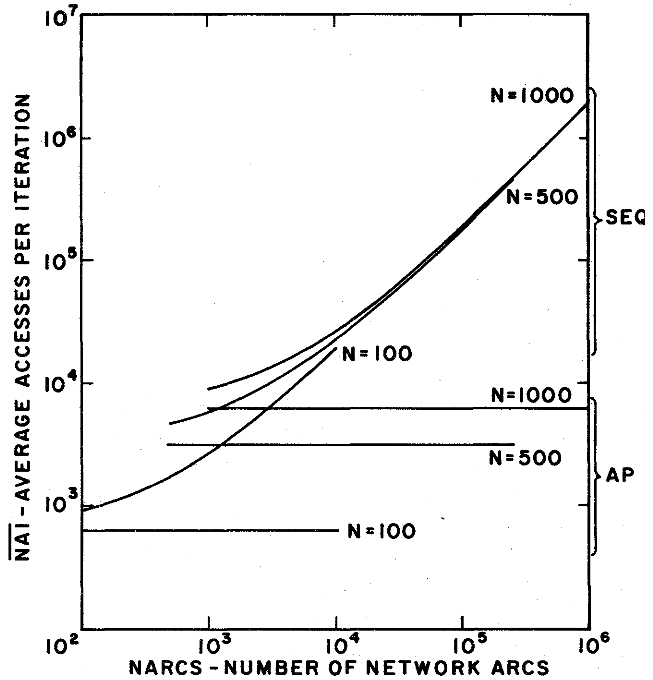


Figure 5—Maximum flow access requirements

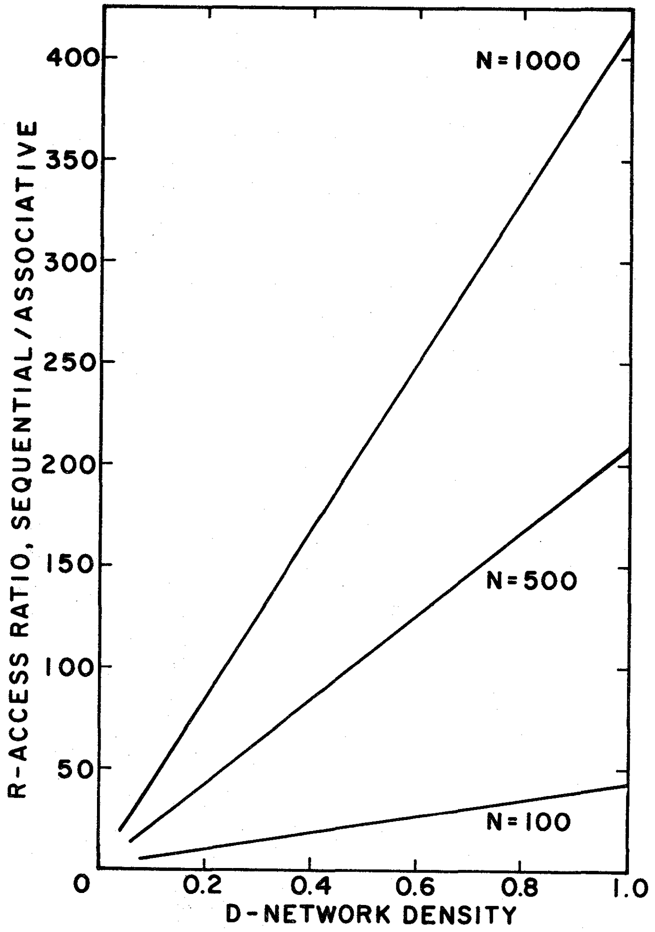


Figure 4—Minimum cost flow access ratio

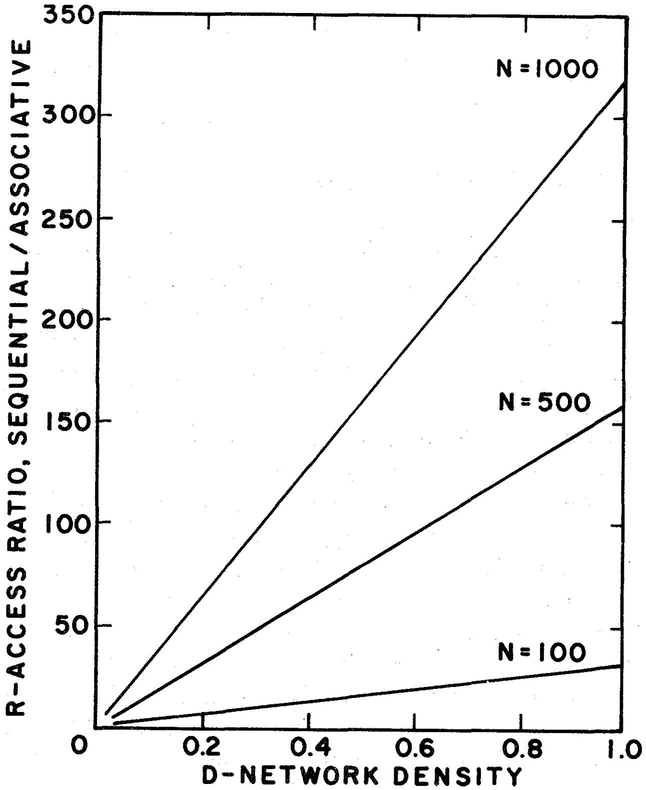


Figure 6—Maximum flow access ratio

definition imposes the constraint a, $b \leq 1$. Making this substitution and forming the ratio of sequential to associative accesses yields

$$R = \frac{NARCS(12+9/N)+N(5+32a+9b)-42}{N(26a+3b)+49}. \quad (14)$$

Since, a, $b \leq 1$esselecting $a=b=1$ gives the most conservative assessment of the impact of the associative processor as applied to this problem. Recall that this implies that $\overline{NLAB}=\overline{NAUG}=N$. Substituting these values into equations (12), (13) and (14) yields

$$\overline{NAI}_{SEQ}=NARCS(6+4.5/N)+23\,N-21 \quad (15)$$

$$\overline{NAI}_{AP}=\tfrac{1}{2}(29\,N+49) \quad (16)$$

$$R=\frac{NARCS(12+9/N)+46\,N-42}{29\,N+49}. \quad (17)$$

The solution of these equations over a representative range of node and arc values results in the data of Table 1 which are presented graphically in Figures 3 and 4. The associative processor access requirements are seen to remain constant with changes in the number of network arcs, reflecting the parallel manner in which the arc data are processed. As shown in Figure 4, the access ratio data of Table I are plotted against network density which is defined as

$$D=\frac{NARCS}{N(N-1)}. \quad (18)$$

Analysis of the preceding data indicates that the access ratio R lies in the range

$$2.0 \leq R \leq 0.4\,N \qquad \text{for} \quad N \geq 100$$

depending upon the density of the network. Because of the approach used, this is an indication of a lower bound on the performance improvement afforded by the associative processor and values of R considerably greater than this bound would typically be expected.

TABLE II—Maximum Flow Access Performance Data

		ASSO-CIA-TIVE	SE-QUEN-TIAL		
N	NARCS	NAI	NAI	R	D
100	100	625	926	1.5	.01
	1,000		1,654	4.2	.10
	6,000		12,254	19.6	.61
	10,000		19,934	31.9	1.00
500	500	3,125	4,726	1.5	.002
	1,000		5,718	1.8	.004
	10,000		23,574	7.5	.04
	100,000		202,134	64.7	.40
	150,000		301,334	96.4	.60
	250,000		499,734	159.9	1.00
1,000	1,000	6,250	9,476	1.5	.001
	10,000		27,404	4.4	.01
	100,000		206,684	33.1	.10
	600,000		1,202,684	192.4	.60
	1,000,000		1,999,484	319.9	1.00

An equivalent analysis[6] for the maximum flow problem yields a sequential program storage requirement of 5(NARCS+1) words against an associative requirement of (NARCS+1) storage words.

Access expressions for this problem were determined to be

$$\overline{NAI}_{SEQ}=NARCS(2-8/N)+7.5\,N-16 \quad (19)$$

$$\overline{NAI}_{AP}=6.25\,N \quad (20)$$

Performance data resulting from these expressions, presented in Table II and Figures 5 & 6, indicate that

$$1.5 \leq R \leq 0.3\,N \qquad \text{for} \quad N \geq 100.$$

SUMMARY

A comparison was made of the relative performance of the associative processor to present sequential computers on the basis of storage requirements for problem data and the number of times that these data were accessed in the course of solving the minimum cost flow and maximum flow problems. It was indicated that the ratio of sequential to associative storage accesses gives an approximate indication of the ratio of execution times to be expected assuming typical hardware speeds for each processor.

Sequential comparison data were obtained through FORTRAN implementation of algorithms published by the ACM as representing typical examples of sequential solutions to these problems. Storage word requirements were obtained directly from the program declarations while access data were obtained by inserting counters to accumulate the number of times that the problem data were accessed in the execution of the sequential programs.

Flow diagrams for the associative processor solution of these problems were developed based upon the capabilities inherent in an associative processor. By analyzing these diagrams it was possible to calculate the number of memory words required for problem data as well as the number of storage accesses required in the execution of the algorithms. To test the correctness of the derived algorithms and verify the accuracy of the access calculations, the algorithms were programmed in associative statements at the assembly language level and executed on an interpretive emulator program written in FORTRAN and run on the Syracuse University Computing Center IBM 360/50. Emulation was required since large scale examples of the associative hardware are not yet available.

It was shown that the storage requirements for the minimum cost flow and maximum flow problems were 7 NARCS+N and 5(NARCS+1) words respectively, where NARCS is the number of arcs and N is the number of nodes in the network. The number of associative processor words was determined to be NARCS+1 in both cases. Considering the differences in word lengths, both systems require approximately the same amount of storage.

The access expressions for each of the competing programs were simplified assuming a best case for the sequential and a worst case for the associative processor. Under the stated assumption, the resulting ratio ranges of

$$\left.\begin{array}{l} 2.0 \leq R \leq 0.4\,N \\ 1.5 \leq R \leq 0.3\,N \end{array}\right\} \text{ for } N \geq 100$$

represent a lower bound on the performance improvement to be expected through the application of the associative processor to the solution of the minimum cost flow and maximum flow problems respectively.

REFERENCES

1 A G HANLON
Content addressable and associative memory systems; a survey
IEEE Transactions on Electronic Computers August 1966 p 509

2 J A RUDOLPH L C FULMER W C MEILANDER
The coming of age of the associative processor
Electronics February 1971

3 A WOLINSKY
Principals and applications of associative memories
Presented to the Third Annual Symposium on the Interface of Computer Science and Statistics Los Angeles California January 30 1969

4 J MINKER
Bibliography 25: An overview of associative or content-addressable memory systems and a KWIC index to the literature: 1956–1970
ACM Computing Reviews October 1971 p 453

5 W L MIRANKER
A survey of parallelism in numerical analysis
SIAM Review Vol 13 No 4 October 1971 p 524

6 V A ORLANDO
Associative processing in the solution of network problems
Unpublished doctoral dissertation Syracuse University January 1972

7 V A ORLANDO P B BERRA
Associative processors in the solution of network problems
39th National ORSA Meeting May 1971

8 CACM
Collected algorithms from the communications of the association for computing machinery
ACM Looseleaf Service

9 T C BRAY C WITZGALL
Algorithm 336 netflow
Communications of the ACM September 1968 p 631

10 D R FULKERSON
An out-of-kilter method for the minimal cost flow problem
Journal of the SIAM March 1961 p 18

11 G BAYER
Algorithm 324 maxflow
Communications of the ACM February 1968 p 117

12 L R FORD D R FULKERSON
Flows in networks
Princeton University Press 1962

13 T C HU
Integer programming and network flows
Addison-Wesley 1969

14 H M WAGNER
Principals of operations research
Prentice-Hall Inc 1969

15 *Manual GER 13738*
Goodyear Aerospace Corporation

Minicomputer models for non-linear dynamic systems

by J. RAAMOT

Western Electric Company, Inc.
Princeton, New Jersey

INTRODUCTION

The computational methods of integer arithmetic have been extended to a variety of applications since the first publication.[1,2] The most noteworthy application of integer arithmetic is the calculation of numerical solutions to initial value problems. This method is introduced here with the example of the differential equation:

$$\frac{dx}{dt}+x=0 \tag{1}$$

By substituting the variable y in place of the derivative, the equation becomes

$$y+x=0 \tag{2}$$

which represents a trajectory in the phase-plane. Given an initial solution point (x_0, y_0, t_0), other solution points (x, y, t) are readily found by first solving the phase-plane equation and then computing the values of the variable t from rewriting the above equation as

$$t=-\int\frac{dx}{x} \tag{3}$$

This example of finding solution points to an initial-value problem demonstrates the procedure which is used in integer arithmetic solutions. Other solution schemes avoid this procedure because in the general case the phase-plane trajectory cannot be expressed in the form $f(x, y)=0$, and an incremental calculation of solution points (x, y) builds up errors.

The major contribution here is that with integer arithmetic techniques, the points (x, y) along the phase-plane trajectory can be calculated with no accumulation of error in incremental calculations, even though the trajectory cannot be expressed in a closed form. As a result, this method handles with equal ease all initial-value problems without making distinctions as to non-linearity, order of differential equation, and homogeneity.

It is necessary to have some understanding of basic integer arithmetic operations before the solution scheme can be discussed. Therefore, the following sections introduce the concepts of F-space and difference terms which are used in integer arithmetic.

F-SPACE SURFACES

A common problem in mathematics is to find the roots of an expression: Given some $f(x)$ the task is to find the values of x which satisfy the equation $f(x)=0$. These roots can be obtained by a method of trial and error where successive values of x are chosen until the equation is satisfied. A simpler method is to introduce the additional variable y, and to find the points on $y=f(x)$ where the contour crosses the x-axis.

This technique of introducing one additional variable is central to operations of integer arithmetic. In the two-dimensional case, a contour $f(x, y)=0$ is the intersection of the surface $F=f(x, y)$ with the xy-plane. Here F is the additional variable and is denoted by a capital letter in order to develop a simple notation for subsequent operations. This three-space is called F-space. It can be created for any dimensionality as is indicated in the table in Figure 1.

Integer arithmetic is not concerned with an analytic characterization of F-space surfaces, but with a set of solution points (F, x, y) at integer points (x, y). The integer points are established by scaling the variables so that unity represents their smallest significant increment over a finite range.

In mathematical calculations the use of integer calculations is avoided because each multiplication may double the number of digits which have to be retained, and the resultant numbers tend to become impractically large. This does not happen in integer arithmetic because the values of F are evaluated at adjacent integer points (x, y) and are expressed as differences. Thereby multiplication is avoided and addition of the differences is the only mathematical operation that is used.

INTEGER ARITHMETIC

DIMENSION	SOLUTION TO EQUATION	F-SPACE EQUATION
1	f(x)=0 ROOTS	y=f(x) CONTOUR
2	f(x,y)=0 CONTOUR	F=f(x,y) SURFACE

	f(x,y)=0
	y=0
	ROOTS
		F1=f(x,y)
		F2=y
		CONTOUR

3	f(x,y,z)=0 SURFACE	F=f(x,y,z) SURFACE

Figure 1—Table of solutions to equations which are obtained by operations in F-space

The F-space solutions are exact for polynomials $f(x, y) = 0$ at all integer points (x, y). Also the F-space surface $F = f(x, y)$ is single-valued in F. Therefore, there is no error in calculating successive solution points based on differences, and the solution at any one point does not depend on the path chosen to get there. These properties do not hold for non-polynomial functions (e.g., exponentials), but there the F-space surface points can be guaranteed to be accurate to within one unit in x and y over a finite range.[1]

DIFFERENCE TERMS

Let the value of the variable F at a point $(x, y, \ldots)$ be F and be $F|_{x+1}$ at $(x+1, y, \ldots)$. Then the first difference term in x is defined by the identity

$$F_x = F|_{x+1} - F \tag{4}$$

It is the change in the F-variable on x-incrementation. This identity can be rewritten for the F-space surface

$$F = f(x, y) \tag{5}$$

as

$$F_x = f(x \pm 1, y) - f(x, y) = \sum_{n=1}^{\infty} \frac{(\pm 1)^n}{n!} \frac{\partial^n f(x, y)}{\partial x^n} \tag{6}$$

The notation F_x is chosen for difference terms in order to distinguish it from finite differences in difference equations, from slack variables in linear programming, and from partial derivative notation, because F_x has a relationship to all of these but differs in its interpretation and use.

The notation for indicating the direction of incrementation is to have F_x or F_{-x}. In addition, the difference between successive difference terms is the second difference term in x and is defined by the identity:

$$F_{xx} = F_x|_{x+1} - F_x \tag{7}$$

In an initial-value problem, the problem is to follow the phase-plane trajectory from an initial starting point. There are various integer arithmetic contour-following algorithms which are based on the sign, magnitude, or a bound on the F-value. To use any one of these algorithms it is necessary to specify the start point (F, x, y), the direction, and the difference terms. To apply these contour-following algorithms to a phase-plane trajectory, the difference terms have to be established. This forms the major portion of the computation.

There are two methods for finding the difference terms and both are illustrated here for the equation of the circle

$$x^2 + y^2 = r^2 \tag{8}$$

It forms the F-space surface

$$F = r^2 - (x^2 + y^2) \tag{9}$$

which is a paraboloid of revolution and is illustrated in

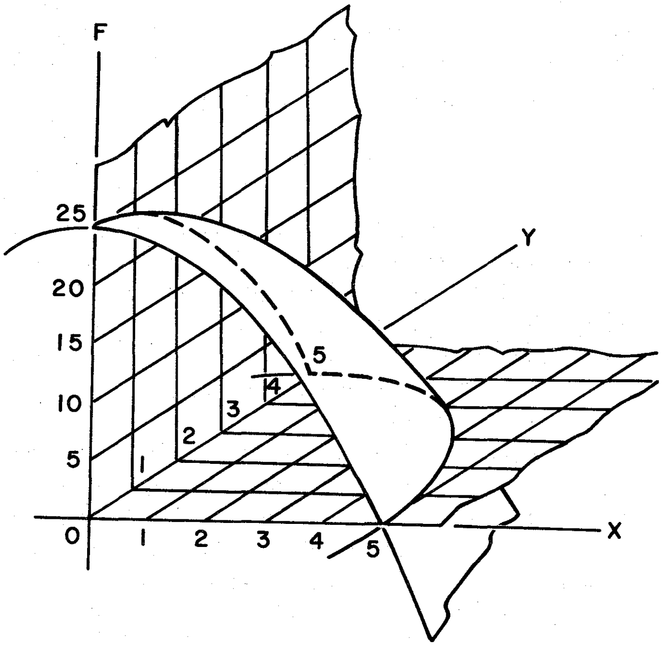


Figure 2—The F-space surface of the circle, $F = 25 - (x^2+y^2)$ is a paraboloid of revolution

Figure 2. The intersection of this surface with the xy-plane forms the circle of radius r.

Case 1: Given $F=f(x,y)$

In this case the difference terms are established from the identity:

$$\begin{aligned} F_{\pm x} &= F\,|_{x\pm 1} - F \\ &= f(x\pm 1, y) - f(x, y) \\ &= \mp(2x\pm 1) \end{aligned} \tag{10}$$

Likewise, it can be demonstrated that the first y difference term is

$$F_{\pm y} = \mp(2y\pm 1) \tag{11}$$

Both difference terms for the circle are illustrated in Figure 3.

Case 2: Given dy/dx

Based on the definition of the derivative, it can be shown that the derivative at a point (x, y) is bounded by the ratio of difference terms at adjacent integer points. Thus, at a point (x, y) the derivative is a good approximation of the ratio of difference terms, and vice versa:

$$\frac{dy}{dx} \simeq \frac{-F_x}{F_y} \tag{12}$$

The exact difference terms are found from this approximation by requiring the F-space surface to be single-valued in F. This requirement results in the identities

$$F_x = -F_{-(x+1)} \tag{13}$$

and

$$F_y = -F_{-(y+1)} \tag{14}$$

which state that the difference in F-values between two adjacent integer points does not depend on the direction.

In the example of the circle, the derivative is

$$\frac{dy}{dx} = -\frac{x}{y} \tag{15}$$

For this given derivative equations (13) and (14) are satisfied only by the introduction of additional terms, here constants c, such that

$$x+c = (x+1) - c \tag{16}$$

and

$$2c = 1 \tag{17}$$

The resultant ratio of the difference terms is

$$\frac{F_x}{F_y} = \frac{2x+1}{2y+1} \tag{18}$$

and can be written as

$$\frac{F_{\pm x}}{F_{\pm y}} = \frac{\mp(2x\pm 1)}{\mp(2y\pm 1)} \tag{19}$$

This result is identical to the one in case 1.

To summarize, in case 2 the function $f(x, y) = 0$ was not given but its derivative was. This is sufficient to calculate the correct difference terms. It is easy enough to verify that incremental calculation of solutions (F, x, y) based on the difference terms $F_{\pm x}$ and $F_{\pm y}$ is exact, accumulates no errors, and represents integer solution points on a single-valued surface in F-space. The reader can also easily verify that the intersection of the F-space surface with the xy-plane is the given circle.

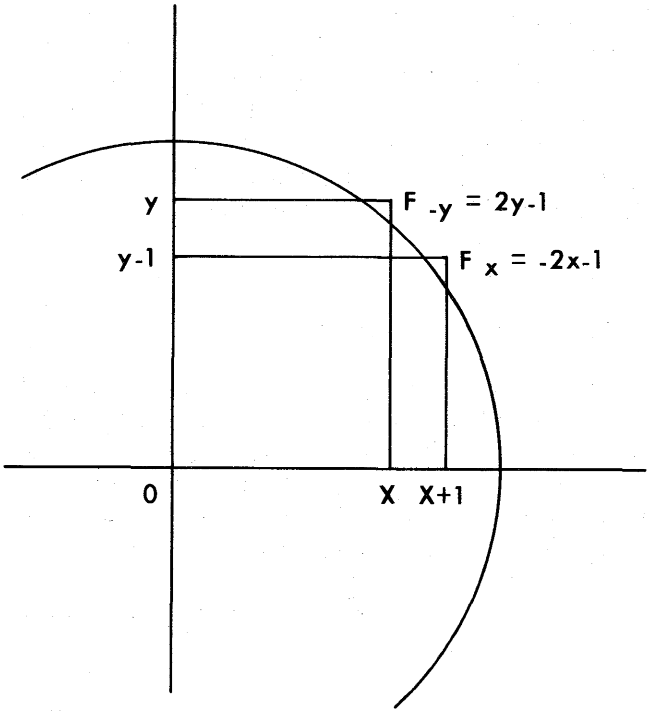


Figure 3—Integer arithmetic difference terms are the changes in the F variable between adjacent integer points in the xy-plane. Successive points are selected to be along the circle but not necessarily on the circle

SECOND ORDER DIFFERENTIAL EQUATIONS

A general second order non-linear differential equation is represented by the equations

$$\frac{dx}{dt}=y \tag{20}$$

$$\frac{dy}{dt}+g(x,y)=0 \tag{21}$$

The first step in finding numerical solutions to the equations is to calculate the difference terms F_x and F_y. Their ratio is approximately

$$\frac{F_x}{F_y}\simeq\frac{-dy}{dx}=\frac{-dy}{dt}\cdot\frac{dt}{dx}=\frac{g(x,y)}{y} \tag{22}$$

which can be written immediately from the above equations.

The exact values of the difference terms must satisfy the identities of equations (13) and (14). These identities are formed by adding appropriate additional terms, $g'(x,y)$ to both sides. The resultant difference term then is

$$F_x=g(x,y)+g'(x,y) \tag{23}$$

In a similar fashion, the difference terms

$$F_y=y+c \tag{24}$$

and

$$-F_{-(y+1)}=(y+1)-c \tag{25}$$

are identical if $c=\frac{1}{2}$. It is not practical to reduce further the ratio of exact difference terms in the general case. Later, specific examples will illustrate this technique.

Given the exact difference terms and the initial values, then any one of the integer arithmetic contour-following algorithms can be applied to find adjacent integer points (x, y) along the phase-plane trajectory without accumulating errors, and the points (x, y) are guaranteed to be accurate to within unity (which is scaled as the least significant increment in the calculations).

Successive solution points are calculated by incrementing one variable in the phase-plane and adding the corresponding difference term to F. In general, the solution points (F, x, y) are on the F-space surface but are not contained in the phase-plane. The important result is that there is no accumulation of errors in the incremental calculation of solution points on the F-space surface.

Errors are introduced in relating the F-space surface points to the phase-plane trajectory, but the contour-following algorithms can always guarantee that these errors are less than unity.

One example of a contour-following algorithm based on the sign is the following: Given an initial direction vector, then the choice of the next increment in the phase-plane is the one which has the difference term sign opposite that from the F-value. If both difference terms and F have the same sign, then the direction is changed to an adjacent quadrant and an increment is taken along the direction axis traversed. Subsequently, the choice of either a positive or negative t-increment determines whether the new direction is acceptable or another change of direction has to be made.

Values of the variable t are obtained by any one of the two following integration steps. Either,

$$t=\int\frac{dx}{y}\simeq\sum_x\frac{1}{F_y} \tag{26}$$

or

$$t=-\int\frac{dy}{g(x,y)}\simeq\sum_y\frac{1}{F_x} \tag{27}$$

will result in the same value of the variable t. Here the integration is approximated by an incremental summation over either x or y increments.

The error in t-increments becomes large whenever the difference term in the denominator of the summation becomes small. This problem is avoided by choosing the summation which contains the largest difference term.

In order for t to increase in the positive sense, the direction of incrementation in the phase-plane is chosen to make the product of the x-direction vector and y-value be positive. Otherwise, t increases in the negative sense. This result is derived from equations (20) through (22). Thereby, the solution method is complete.

EXAMPLE 1

The integer arithmetic method of finding numerical solutions to differential equations is illustrated here by the example of a second order linear differential equation. It has easily derivable parametric solutions in t but its phase-plane trajectory cannot be expressed as $f(x,y)=0$. This initial value problem is stated as

$$\frac{dx}{dt}=y \tag{28}$$

$$\frac{dy}{dt}+ky+w^2x=\dot{y}_0 \tag{29}$$

with initial values of

$$(x_0, y_0, \dot{y}_0)=(0, 20, 0) \tag{30}$$

Its parametric solution depends on the values chosen for the constants k and w^2. The values chosen here are $k=.08$ and $w^2=.04$ which result in the parametric solution

$$x=104.9e^{-.04t}\sin 0.196t \qquad (31)$$

and

$$y=-4.19e^{-.04t}\sin 0.196t+20.0e^{-.04t}\cos 0.196t \qquad (32)$$

These calculations apply only for the analytic solution and not in the integer arithmetic solution scheme. There, the first step is to establish the difference terms from the given equations (28) and (29).

According to equation (12) the approximate ratio of the difference terms is

$$\frac{F_x}{F_y}\simeq\frac{ky+w^2x-\dot{y}_0}{y} \qquad (33)$$

The requirement that the F-space surface is a single-valued surface, as stated in equations (13) and (14), is applied to obtain the exact difference terms

$$F_x=ky+w^2(x+\tfrac{1}{2})-\dot{y}_0 \qquad (34)$$

and

$$F_y=y+\tfrac{1}{2} \qquad (35)$$

Then the ratio of the terms is multiplied by $2n/2n$ where n is an appropriate integer to eliminate the fractions. The choice of direction in the phase-plane for positive t establishes that the x, y-direction vector is $(1, -1)$, and the difference terms are:

$$F_{\pm x}=\pm n[2ky+w^2(2x\pm 1)] \qquad (36)$$

$$F_{\pm y}=\pm n(2y\pm 1) \qquad (37)$$

The resultant phase-plane trajectory is illustrated in Figure 4 and the numerical results (x, t) are compared with the calculation of values of x in real time in Figure 5. The peak to peak values of x are equal to 114 increments which corresponds to a 1 percent accuracy. As can be seen, the first cycle is calculated in 25 milliseconds. A comparison of the numerical with the analytic solution confirms that all points (x, y) are unit distant from the true trajectory.

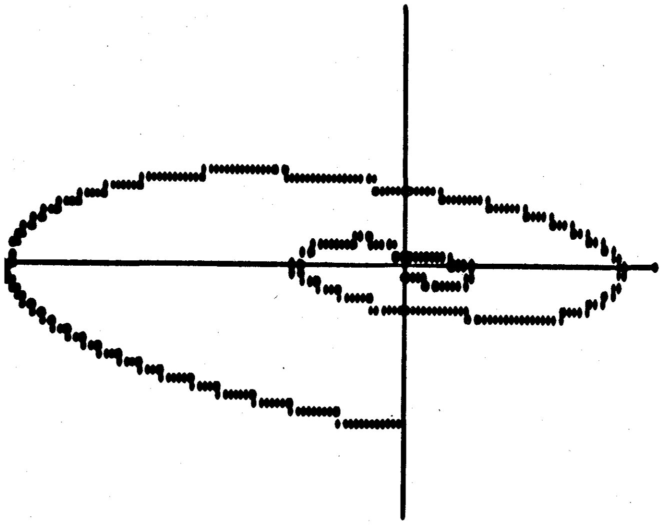


Figure 4—The phase-plane trajectory of the linear differential equation discussed in example 1, for the initial values (x, y, t) = (0, 20, 0). The integer arithmetic solutions are displayed on a CRT

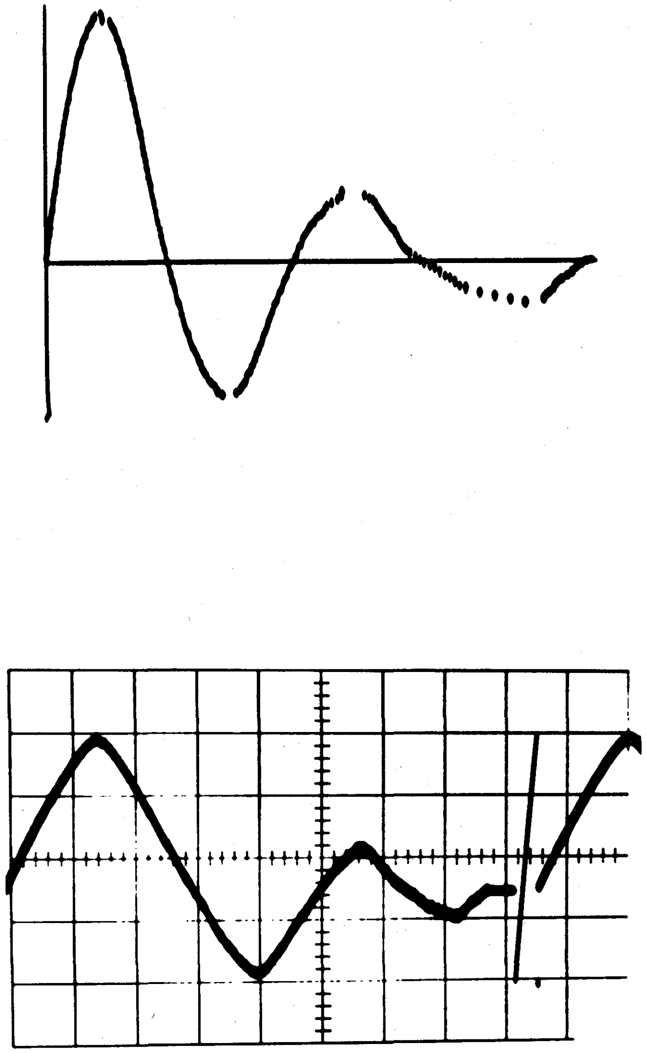


Figure 5—Top: The integer arithmetic solutions (x, t) as displayed on a CRT for the trajectory of Figure 4
Bottom: A CRT display of the integer arithmetic solutions x as calculated in real time. The time axis is 5msec/cm

The above example illustrates how the incremental calculations are set up for the second order differential equation. The numerical solutions (x, y, t) are obtained by application of the integer arithmetic calculation, even though there is no closed expression $f(x, y)=0$ for the trajectory.

SCALING

In many problems it is necessary to scale the variables to obtain either an improved or a coarser resolution. Such scaling is best illustrated by the example of the circle given in equation (9).

First, an improved resolution in x only is obtained by taking increments of $1/n$ units where n is an integer and integer calculations are retained. Then,

$$\begin{aligned} F_{nx} &= -[(x+1/n)^2-x^2] \\ &= -[n^{-2}(nx+1)^2-x^2] \\ &= -[n^{-2}(2nx+1)] \end{aligned} \tag{38}$$

On multiplying F by n^2, the difference terms are

$$F_{nx}=-(2nx+1) \tag{39}$$

and

$$F_y=-n^2(2y+1) \tag{40}$$

The other scaling example takes n increments in x at a time. Then,

$$\begin{aligned} F_{x/n} &= -[(x+n)^2-x^2] \\ &= -[n^2(x/n+1)^2-x^2] \\ &= -n^2(2x/n+1) \end{aligned} \tag{41}$$

and the y difference remains

$$F_y=-(2y+1) \tag{42}$$

The last step in scaling is to substitute a new variable for nx or x/n respectively in the above examples, and to proceed with integer calculations.

EXAMPLE 2

The earlier example of the integer arithmetic solution scheme represented a linear differential equation with parametric solutions. For this example is chosen the van der Pol equation

$$\frac{d^2x}{dt^2}+c(x^2-1)\frac{dx}{dt}+x=0 \tag{43}$$

The phase-plane trajectory of this equation has a stable limit cycle with a radius of 2 for the constant $c>0$. Near the limit cycle, it is necessary to scale the problem to obtain an improved resolution of integer solution points. This is done by rewriting the van der Pol equation with the new variables x' and y'

$$x=x' \tag{44}$$

$$\frac{dx}{dt}=y' \tag{45}$$

and by taking increments of $1/n$ units in x', $1/m$ units in y', and replacing c by c/k. Then the difference terms are computed, and the variable nx' is replaced by x and my' is replaced by y. The resultant difference terms are:

$$F_{\pm x}=m[\pm c(6x^2\pm 6x+2-6n^2)y+nmk(\pm 6x+3)] \tag{46}$$

and

$$F_{\pm y}=n^3k(\pm 6y+3) \tag{47}$$

For $m=n=10$ and $c=k=1$, the point $(x, y, F)=(0, 21, -83840)$ is located in F-space on the limit cycle F-space surface. Based on the difference terms, an incremental calculation of solution points along the limit cycle returns to the same start point. This confirms that there is no accumulation of errors in the incremental calculations.

Likewise, for start points chosen both inside and outside the limit cycle, results agree with the expected trajectories in that all resultant trajectories terminate with the limit cycle. These results are in complete agreement with published data[3] for values of the constant $c=0.1$, 1.0, and 10.

The variable t is calculated from either summation:

$$t=\sum_x \frac{K}{F_y} \tag{48}$$

or

$$t=\sum_y \frac{K}{F_x} \tag{49}$$

It should be remembered that x, y, and F have been rescaled and correspondingly also the numerator in these summations is scaled to K. It is given by the equation

$$K=6n^2mk \tag{50}$$

for both the x and y incrementation sum.

If the forcing function, $5\sin 2.5t$ is applied to the van der Pol equation, then the difference term in x becomes

$$\begin{aligned} F_{\pm x}=m[&\pm c(6x^2\pm 6x+2-6n^2)y+mnk(\pm 6x+3) \\ &\mp 6mn^2k(5\sin 2.5t)] \end{aligned} \tag{51}$$

and the y difference term remains unchanged. In this instance, again the results are in complete agreement with published data.[4]

HIGH-ORDER SYSTEMS

An initial value problem can be written as:

$$\frac{dx1}{dt} = f1(x1, x2, \ldots, xn, t)$$

$$\frac{dx2}{dt} = f2(x1, x2, \ldots, xn, t)$$

$$\vdots$$

$$\frac{dxn}{dt} = fn(x1, x2, \ldots, xn, t) \tag{52}$$

The numerical solutions $(x1, x2, \ldots, xn, t)$ are found by taking the set of equations in the ratios:

$$\frac{dx2}{dx1}, \frac{dx3}{dx2}, \ldots \tag{53}$$

Each ratio represents a phase-plane trajectory for which the difference terms can be established.

An increment in $x2$ in the first phase-plane trajectory also corresponds to an increment in $x2$ in the second phase-plane trajectory, which in turn may result in $x3$ being incremented. Whether or not $x3$ is incremented depends on the particular integer arithmetic contour-following algorithm which is used. For example, the algorithm based on the sign forces an $x3$ increment when the value of the difference term for that variable has the opposite sign of the current F-value.

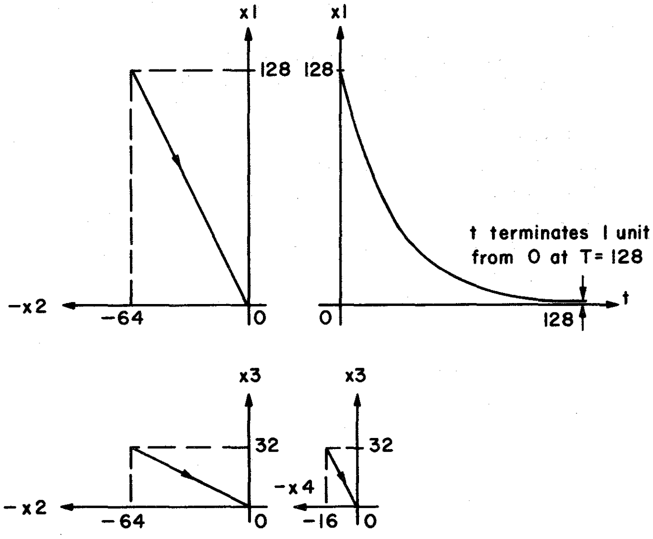
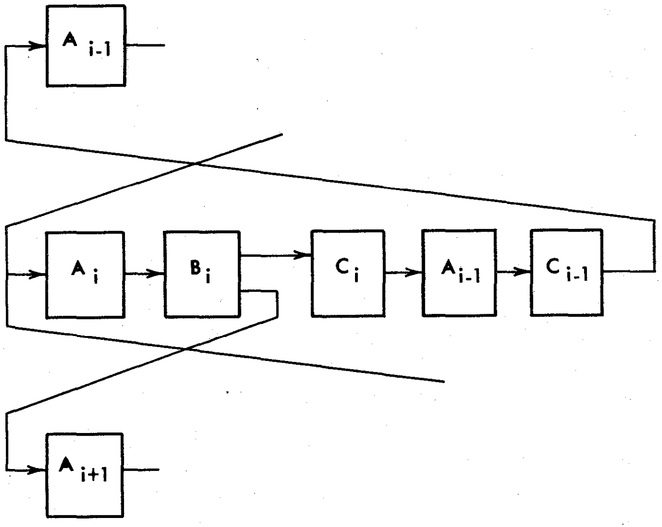


Figure 6—Coupled phase-plane trajectories for the equation,

$$\frac{d^4x}{dt^4} - \frac{1}{8}x = 0$$

The variable t is obtained from the first trajectory and is scaled to $T = 128$ at $x1 = 1$ unit from termination



SUBROUTINE A_i calculates F_{xi} and $F_{x(i+1)}$.

SUBROUTINE B_i chooses next increment or operation.

SUBROUTINE C_i updates all variables for an X_i increment.

Figure 7—Flow chart of an integer arithmetic algorithm for tracking coupled trajectories, showing the calculations for the i-th variable

These coupled trajectories are illustrated in Figure 6 for the simple equation

$$\frac{d^4x}{dt^4} - \frac{1}{8}x = 0 \tag{54}$$

given the initial values $(x1, x2, x3, x4, t) = (128, -64, 32, -16, 0)$. A general algorithm for coupled trajectories is shown in the block diagram of Figure 7. As can be seen, an increment in the first variable may immediately ripple through to an incrementation in the n-th variable, after which, starting from the end of the chain, each trajectory achieves a stable solution as determined by the contour following algorithm.

The variable t can be calculated from any one phase-plane trajectory each resulting in the same value but being consistent with the resolution of computation.

CONCLUSIONS

The integer arithmetic solution method has been applied to a variety of initial-value problems, of which representative examples are illustrated above. Associated with this method are a number of theorems. These prove that the F-space surface is single-valued in F, that the direction field is bounded by the ratio of difference terms, that some trajectories have integer F-space solutions at all integer points in the phase-plane, and that for other trajectories the F-space surface is approximated, but the accuracy of results is guaranteed over a finite domain. However, additional theorems remain to be developed to insure that the method is applicable to all initial-value problems, and to determine the necessary conditions for stability.

The solution method is summarized as follows: Successive solution points along a phase-plane trajectory are calculated by adding a difference term to the F-value and incrementing the associated phase-plane variable. These simple operations are offset by the more complex contour-following algorithms which track the trajectory by examining the state of calculations and then selecting the next increment. Here the underlying concept is that the trajectory is the contour formed by the intersection of the F-space surface with the phase-plane.

There exists a duality between the integer arithmetic technique and the standard Runge-Kutta or predictor-corrector solution methods. In integer arithmetic, the phase-plane variables are the independent variables and t is a dependent variable obtained as a result of integration. Just the reverse is true in the standard methods; t is an independent variable and the phase-plane variables are obtained as a result of integration. The integer arithmetic technique finds solution points on the phase-plane trajectory even though there may not exist an analytical expression of that trajectory. Likewise, the standard method finds solution points of integrals which cannot be expressed in analytic form. An additional duality is that after initial scaling, the integer arithmetic solutions have a guaranteed accuracy whereas the standard methods require a subsequent accuracy calculation.

The computations involved in the integer arithmetic method are simpler than the ones in other methods: The examples illustrated in this paper were programmed in assembly language for the Digital Equipment Corporation PDP-15 computer. It has only 4096 words of store and does not have a hardware multiplier. The entire program is contained in 300 words of store and is executed in 50 microseconds per increment in x or y, including the time calculation. It is difficult to execute any other solution scheme within such limited facilities or comparable speed.

Other examples have been programmed in FORTRAN on a large PDP-10 computer. There the execution time is 10 times slower, and is comparable to the standard numerical integration methods. In these examples, floating point calculations were used for the integer arithmetic calculations.

ACKNOWLEDGMENTS

The development of this method resulted from the application of integer arithmetic techniques at the Western Electric Engineering Research Center in Princeton. Also, there are substantial contributions by J. E. Gorman in formulating the integer arithmetic techniques.

REFERENCES

1 J E GORMAN J RAAMOT
Integer arithmetic technique for digital control computers
Computer Design Vol 9 No 7 pp 51-57 July 1970

2 A G GROSS et al
Computer systems for pattern generator control
The Bell System Technical Journal Vol 49 No 9
pp 2011-2029 November 1970

3 L BRAND
Differential and difference equations
Wiley 1966 New York

4 L LAPIDUS R LUUS
Optimal control of engineering processes
Blaisdell Waltham 1967

Fault insertion techniques and models for digital logic simulation

by STEPHEN A. SZYGENDA and EDWARD W. THOMPSON

Southern Methodist University
Dallas, Texas

INTRODUCTION

During the past few years it has become increasingly apparent that in order to design and develop highly reliable and maintainable digital logic systems it is necessary to be able to accurately simulate those systems. Not only is it necessary to be able to simulate a logic net as it was intended to behave, but it is also necessary to be able to model or simulate the behavior of the logic net when it contains a physical defect. (The representation of a physical defect is known as a fault.) The behavioral simulation of a digital logic net which contains a physical defect, or fault, is known as digital fault simulation.[1-6]

In the past, two methods have been used to determine the behavior of a faulty logic net. The first approach was manual fault simulation.[7] (For logic nets of even moderate size, this method is slow and often inaccurate.) The second method used is physical fault insertion.[7] In this method faults are physically placed in the fabricated logic, input stimuli are applied and the behavior of the logic net observed. Although physical fault insertion is more accurate than manual fault simulation, it is still a lengthy process and requires hardware fabrication. The most serious limitation, however, is that physical fault insertion is dependent on a fabrication technology which permits access to the input and output pins of logic elements such as AND gates and OR gates. With discrete logic this is possible, however, the use of MSI and LSI precludes the process of physical fault insertion. Since MSI and LSI will be used in the future for the majority of large digital systems, the importance of digital fault simulation can be readily observed.

The major objective of digital fault simulation is to provide a user tool by which the behavior of a given digital logic design can be observed when a set of stimuli is applied to the fabricated design and a physical defect exists in the circuit. This tool can then be used to validate fault detection or diagnostic tests, to create a fault dictionary, to aid in the automatic generation of diagnostic tests, or to help in the design of diagnosable logic.

The activities of a digital fault simulation system can be divided into two major areas. The first is the basic simulator, which simulates the fault free logic net. The activities of the second part are grouped under the heading of program fault insertion. (For digital fault simulation as opposed to physical fault insertion.)

The merit of the fault insertion activities can be judged on five points. These are:

(1) Accuracy with which faults can be simulated.
(2) Different fault models that can be accommodated.
(3) Methods for enumerating faults to be inserted.
(4) Extraction of information to be used for fault detection or isolation.
(5) Efficiency and capability of handling large numbers of faults.

ACCURACY OF FAULT SIMULATION

In order to accurately predict the behavior of a logic net which contains a fault, the basic simulation used must be capable of race and hazard analysis. A simple example of this is shown in Figure 1. In this example an AND gate has three inputs, a minimum delay of 3, and a maximum delay of 4. At time T_2 signal A starts changing from 1 to 0 and at the same time signal B starts changing from 0 to 1. The period of ambiguity for the signals is 3. For the fault free case, signal C remains constant at 0. Therefore, the output of the gate remains constant regardless of the activity on signals A and B. If signal C has a stuck-at-logical 1 fault, there is potential for a hazard on the output of the gate between time T_5 and T_9. This hazard will not

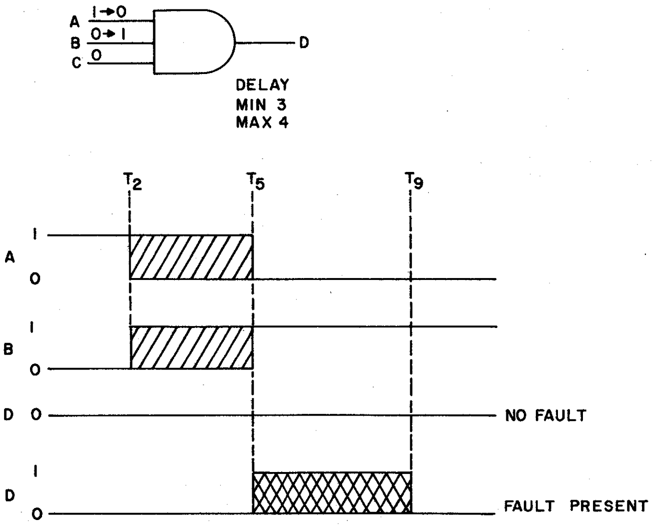


Figure 1—Fault induced potential error

be seen unless the fault insertion is done in conjunction with simulation that is capable of detecting such a hazard.

The fault insertion to be discussed here is done in conjunction with the TEGAS2[8,9] system. TEGAS2 is a table driven assignable delay simulator which has eight basic modes of operation of which three are concerned with faults. For the first mode, each element type is assigned an average or nominal propagation delay time. This is the fastest mode of operation, but it performs no race or hazard analysis. Mode 2 is the same as mode 1, except it carries a third value which indicates if a signal is indeterminate. The third mode has a minimum and maximum propagation delay time associated with each element type and performs race and hazard analysis. All three modes can use differing signal rise and fall times.

Fault insertion and parallel fault simulation are performed in all three of these modes. When fault insertion is done in mode 3, races or hazards that are induced because of a fault, will be detected. If fault insertion is done in mode 2, no fault will be declared detected unless the signal values which are involved in the detection are in a determinate state. Also it can be determined if a fault prevents a gate from being driven to a known state.

By using TEGAS2 as the basic simulator, faults can be simulated to whatever degree of accuracy desired by the user.

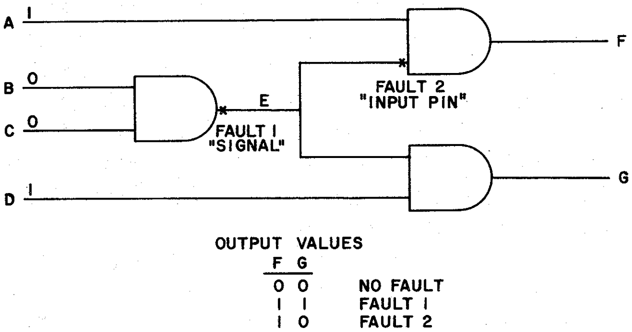
FAULT MODELS

In order for a fault simulation system to be as flexible and as useful as possible, it should be able to model or insert various kinds of faults. Most fault simulation systems are capable of modeling only the class of single occurring stuck-at-logical 1 and stuck-at-logical 0 pin faults. Although it has been found that this class of faults covers a high percentage of all physical defects which occur, (considering present technology), it is certainly not inclusive.

In an effort to remain as flexible and as efficient as possible and to be able to model different classes of faults, TEGAS2 has developed three different fault insertion techniques. The first technique is used to insert signal faults or output pin faults. This type of fault is where an entire signal is stuck-at a logical 1 or a logical 0. The distinguishing factor is that the fault affects an entire signal and not just an input pin on an element. Figure 2 illustrates a signal or output pin fault as opposed to an input pin fault.

At the beginning of a fault simulation pass, the OUTFLT table (Figure 3), which contains all of the signal faults to be inserted, is constructed. There is one row in the table for every signal that is to be faulted. The information in each row is the signal number, MASK1, and MASK2. The signal number is a pointer into an array CV where the signal values are stored. MASK1 has a 1 in each bit position that is to be faulted. The right most bit of a word containing a signal value is never faulted since that bit represents the good machine value. MASK2 contains a 1 in any bit position that is to have a SA1 inserted and a 0 where there is either a SA0 or no fault to be inserted. Parallel fault simulation is accomplished by having each bit position, in a computer a word containing a signal value, represent a fault.

At the end of each time period during simulation, for which any activity takes place, the signal faults are inserted. This method is very simple and requires little extra code. For example, let CV be the array containing the signal values and OUTFLT be the two dimensional table discussed above. Then, to insert one



OUTPUT VALUES

F	G	
0	0	NO FAULT
1	1	FAULT 1
1	0	FAULT 2

Figure 2—Example of a signal fault and a pin fault

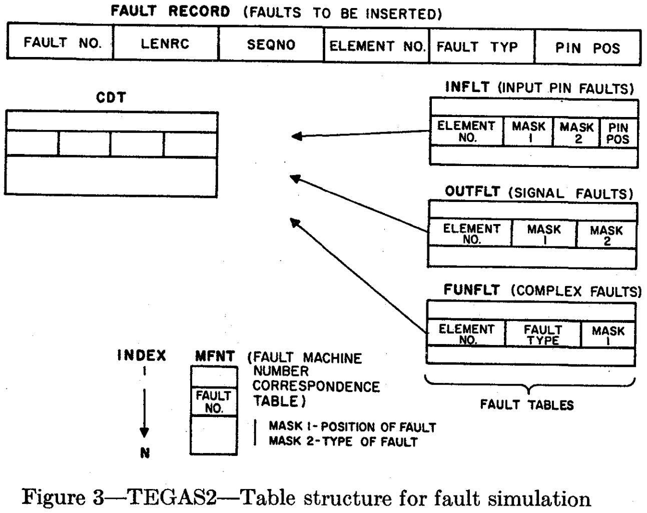


Figure 3—TEGAS2—Table structure for fault simulation

or more faults on a signal, the following statement would be executed.

```
CV(OUTFLT(i,1))  =  (CV(OUTFLT(i,1)).AND.
                    (.NOT.OUTFLT(i,2)).
                    OR.OUTFLT(i,3).[1]
```

i represents the row index in the signal fault table OUTFLT. It is not necessary to insert signal faults after a time period that has no activity, since none of the signals will have changed value. By inserting signal faults in this manner it is not necessary to check a flag every time an element is evaluated to see if its output must be faulted.

The second method of fault insertion is used for input pin faults. An input pin fault only affects an input connection on an element. This is demonstrated in Figure 2. In the table structure for the simulator, each element has pointers to each of its fan-ins. The pointer to any fan-in that is to be faulted is set negative prior to a simulation pass. During simulation the input signals for an element are accessed through a special function. That is, the evaluation routines for the different element types are the same as when fault insertion is not performed except that the input values for the element are acquired through the special function. This function determines if a particular fan-in pointer is negative. If a pointer is negative, the element being evaluated and the signal being accessed are looked up in a table containing all input pin faults to be inserted for a simulation pass. The appropriate fault can then be inserted on the input pin before the evaluation routine uses it.

The input pin faulting procedure can be more clearly illustrated by first examining the major tables used in simulation. These are given in Figure 4. Each row in the circuit description table (CDT) characterizes a signal or a single element. The first entry in the CDT table is a row index into the function description table (FDT). The second entry CDT (i,2), points to the first of the contiguous set of fan-in pointers (in the FI array) for element i. CDT (i,3) points to the first of a contiguous set of fan-out pointers (in the FO array) for element i, and CDT (i,4) specifies how many fan-outs exist. The signal value (CV) table contains signal values. The ith entry in the CV array contains the value of the ith signal or element in the CDT table.

Each row in the FDT table contains information which is common to all elements of a given logical type. FDT (i,1) contains the number of the evaluation routine to be used for this element type, FDT (i,2), FDT (i,5), and FDT (i,6) contain the nominal, maximum, and minimum delay times respectively. FDT (i,3) specifies the number of fan-ins for this element type. FDT (i,7) contains the number of outputs for the given element type. (This is used in the case of multiple output devices.) FDT (i,4) is used for busses.

The simplest evaluation routine for a variable input AND gate will now be given. This is the routine used when no race and hazard analysis is performed, nor is an indeterminate value used.

```
          N = (FDT(CDT(I, 1), 3)
      IFIPT = CDT(I,2)
      ITEMP = ALLONE
   DO 10 NN = 1,N
          K = FI(IFIPT + NN − 1)
      ITEMP = ITEMP.AND. CV(K)
10 CONTINUE
```

The integer variable ALLONE has all bits set to one.

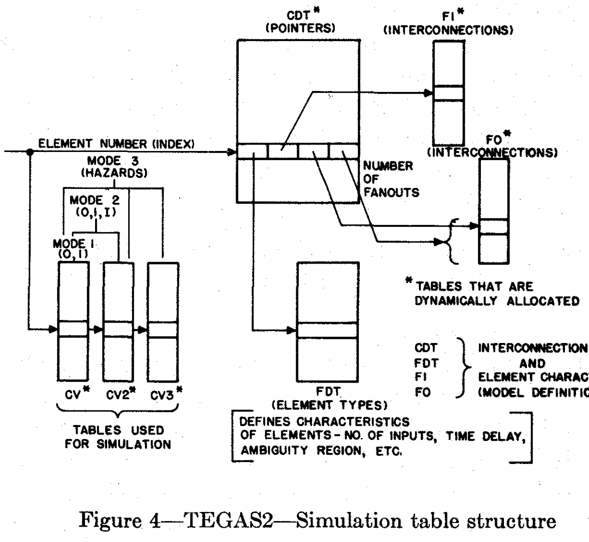

CDT*
(POINTERS)
FI*
(INTERCONNECTIONS)
FO*
(INTERCONNECTIONS)
ELEMENT NUMBER (INDEX)
MODE 3
(HAZARDS)
MODE 2
(0,1,I)
MODE 1
(0,1)
NUMBER OF FANOUTS
*TABLES THAT ARE DYNAMICALLY ALLOCATED
CV*
CV2*
CV3*
TABLES USED FOR SIMULATION
FDT
(ELEMENT TYPES)
CDT
FDT
FI
FO
INTERCONNECTION DATA AND ELEMENT CHARACTERISTICS (MODEL DEFINITION)
DEFINES CHARACTERISTICS OF ELEMENTS - NO. OF INPUTS, TIME DELAY, AMBIGUITY REGION, ETC.

Figure 4—TEGAS2—Simulation table structure

All that is required to change this routine so that input pin faults can be inserted is to replace CV(K) with FLTCV(K). FLTCV is a function call. It determines if the fan-in pointer K is negative and if so it uses the INFT table to insert the appropriate fault. The INFLT is the same as the OUTFLT table except that the relative input pin position, to be faulted, is given. The combination of the element number and the input pin position on that element identify a particular pin to be faulted.

The third method of fault insertion is used for complex faults. A complex or functional fault is a fault used to model a physical defect which does not correspond to a single or multiple occurring stuck-at-logical 1 or stuck-at-logical 0 fault. An example of this is a NAND gate that becomes an AND in the presence of some physical defect. For this approach an element is first evaluated by its normal evaluation routine. Then, if a complex fault is to be inserted on that element, it is evaluated again using a routine which models the complex fault. An example would be an inverter gate which no longer inverts. In this case, the normal inverter routine would be used first, then an evaluation routine, which merely passes the input signal along, would be used.

As with the other insertion techniques, a table (FUNFLT, Figure 3) is constructed at the beginning of a simulation pass and it contains all elements that are to have complex faults for that pass. This table also contains the routine number that will evaluate a prospective complex fault and again which bit position will represent the fault. Each entry in the FUNFLT table has one extra space, used for input pin position when modeling input shorted diodes. It is the responsibility of the complex fault evaluation routines to merge their results with the results of the normal evaluation routine so that the proper bit represents the fault. This is accomplished by using MASKI in the FUNFLT table. Assume that variable SP temporarily contains the non-fault element evaluation results and the variable SPFT contains the results from the element representing the complex fault. As was stated before, MASK1 contains a 1 in the bit position that is to represent the fault, then the statement

SP = (SP.AND. (.NOT.MASK1)) .OR. (MASK1.AND.SPFT)

will insert the fault in SP.

Other faults that can be modeled with the complex fault insertion technique are shorted signals, shorted diodes, NAND gates that operate as AND gates, edge triggered D type flip-flops which are no longer edge triggered, etc. Two signals which are shorted together would be modeled as in Figure 5. A dummy gate is placed over signals A and B. In the faulted case, the dummy gate takes on the function of an AND gate or an OR gate, depending on technology. In this case, the input signals are ANDed or ORed together and the result passes on to both A* and B*.

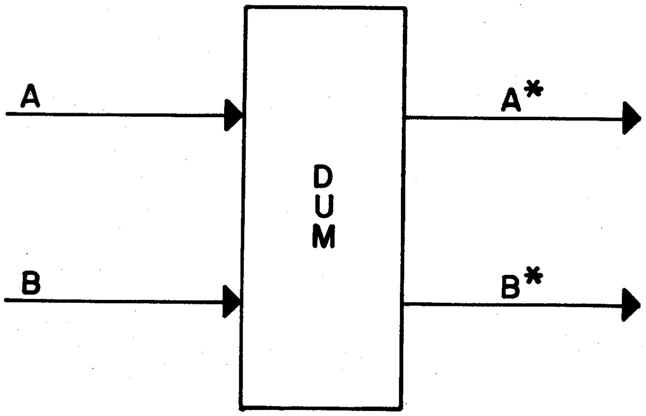


Figure 5—Shorted signals

Another class of faults that can be modeled, to some extent, with the complex method, is transient or intermittent faults. This is possible only because TEGAS2 is a time based simulator. As an example, let us model the condition of a particular signal periodically going to 0 independent of what its value is supposed to be. Again we pass the signal through a dummy gate as in Figure 6. The dummy gate also produces the fictitious signal (F) which is ANDed with the normal signal. The fictitious signal is normally 1, however the dummy gate can use a random number generator to periodically schedule the fictitious signal to go to 0. It can also use the random number generator or a given parameter to determine how long the signal remains at 0. From this discussion, the flexibility and power of the complex fault insertion method can be seen.

In addition to modeling the faults described above, any combination of faults can be modeled as if they existed simultaneously. A group of faults that exist at the same time is considered to be a multiple fault. With this capability multiple occurring logical stuck at 1 and logical stuck at 0 faults can be modeled. Also multiple complex faults can be inserted, or any combination of the above.

Modeling a group of multiple faults is accomplished simply by letting a single bit position in the signal values represent each of the faults that are to exist together. That is, MASK1 in the fault tables would have the same bit position set to one for each of the faults in a group of multiple occurring faults.

This approach to the handling of multiple faults has permitted us to develop a new technique for simulating any number of faults, from one fault to all faults, in one simulation pass. The added running time for this approach is slightly more than that needed to do a parallel simulation pass which consideres a number of faults equal to the host machine word length minus one. Hence, the approach has the potential of being less time consuming than the one pass simulators[10] and more flexible and efficient than the traditional parallel simulators. The technique is called *M*ultiple *N*umber of *F*aults/*P*ass (MNFP) and partitions the faults into classes that will be simulated as multiple faults. Therefore, each bit position represents a group of faults. If the groups are structured such that blocking faults (such as a stuck at 1 and a stuck at 0 simultaneously on the same signal) are not included in the same group, fault detection can be achieved. If fault isolation is required, the fault groups which contain detected faults will be repartitioned and the process continued. For example, if we are simulating 35 groups of faults and five groups indicate faults being detected, the five groups will be repartitioned and simulated for isolation. The efficiency of this approach is derived from the fact that the other 30 groups need not be simulated any further for these inputs. Assume that these 30 groups each contained 70 faults. For parallel simulation, this would require 59 additional simulation passes, over the (MNFP) approach.

Another feature of this approach is that all faults need not, and indeed, sometimes should not, be simulated in one pass. For example, assume the following partition of 2,168 faults.

Number of groups	Number of faults/group
10	19
5	27
20	35
15	71
10	.6
9	2
—	—
69 (Total Groups)	2,168 (Total Number of Faults)

For this case, two passes of the simulator would be required, assuming a 36 bit host machine word.

MNFP is also being used in conjunction with diagnostic test generation. For example, assume the existence of 3500 faults, and that our diagnostic test generation heuristics have generated three potential tests (T_1, T_2 and T_3). If the faults are partitioned into groups of 100 each, all faults could be simulated in one pass. Hence, if each test is applied to the fault groups using MNFP, it would require three passes to determine (to some degree) the relative effectiveness of the tests. If T_1 and T_2 detected faults in only one group, and T_3 detected faults in 5 groups (with fault detection being the objective), the most likely candidate for further analysis would be T_3. Even if all of the faults in these 5 groups were then considered individually, (the worst case) the entire process would require 18 simulation passes, as opposed to 100 passes using conventional parallel simulation. Further studies to determine the most efficient utilization of the MNFP technique are presently under way.

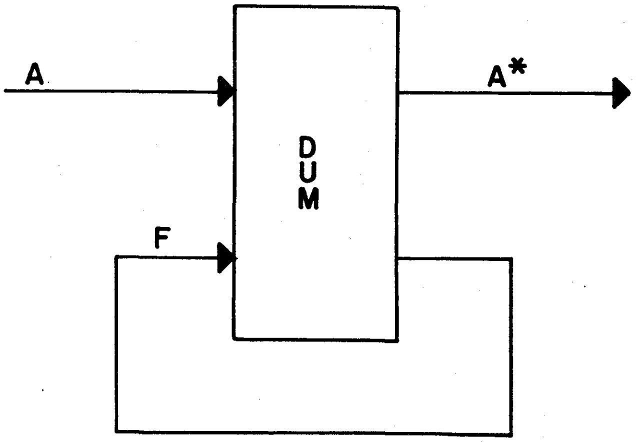


Figure 6—Intermittent fault

ENUMERATION OF FAULTS

If every fault that is to be inserted must be specified manually, it could be a very laborious process. It is certainly necessary to be able to specify faults manually if desired, but it is also necessary to be able to generate certain classes of faults automatically. TEGAS2 is set up in a modular fashion such that control cards can be used to invoke independent subroutines which will generate different classes of faults. Additional faults can be specified manually. New subroutines can be easily added to generate new classes of faults as the user desires. One class of faults that the system presently generates automatically, is a collapsed set of single occurring stuck-at-logical 1 and stuck-at-logical 0 pin faults. This is the class of faults most often used.

A collapsed set of faults refers to the fact that many faults are indistinguishable. For instance, a stuck at 0 on any of the input pins of an AND gate and a stuck at 0 on the output of the AND gate cannot be distinguished. If this group of faults is collapsed into one representative fault, it is considered to be a simple gate collapse. This is easy to perform and many existing systems utilize this feature.

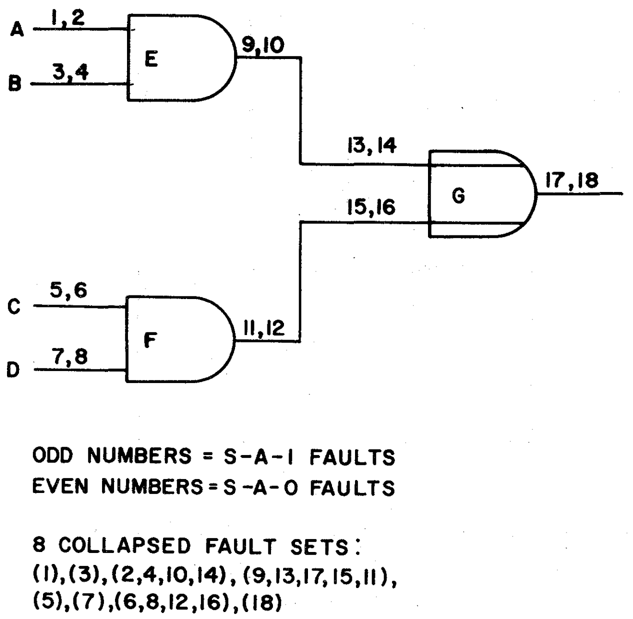


Figure 7—Fault collapse

A simple gate collapse is not, however, a complete collapse. Figure 7 gives an example of a completely collapsed set of faults. There are a total of 18 possible single occurring S-A-1, S-A-0 faults. A simple gate collapse will result in 12 sets of faults. However, an extended collapse results in only 8 sets of distinguishable faults. This amounts to a reduction of 33 percent over the simple collapse. In large examples, the extended co-l lapse has consistently shown a reduction of approximately 35 percent over the simple collapse.

The information gained in collapsing a set of faults has additional value. This information can be used in determining optimal packaging of elements in MSI or LSI arrays so as to gain fault resolution to a replaceable module. As in Figure 7, it can be readily seen that these three elements might best be placed on the same replaceable module. This is true because all three elements are involved in an indistinguishable set of faults.

FAULT DETECTION

The activity associated with determining when a fault has caused an observable malfunction can be termed fault detection. As with many other functions, TEGAS2 uses a dummy gate designated as the detection gate, for this purpose. A detection gate is specified for signals that are declared observable for the purpose of fault detection. These signals are many times referred to as primary outputs or test points. An ordered set of such signals is called the output vector. If any fault causes the value of one or more points on the output vector to be different from when the fault is not present, the fault is declared detected.

Whenever one of the signals, which is part of the output vector, changes value, its corresponding detection gate determines if a fault is observable at that point. This is easily accomplished since the fault free value for any signal is always stored in the low order bit of the host machine word. The values for that signal, corresponding to each of the faults being simulated at that time, are represented by the other bits in the word. Hence, all that is necessary is to compare each succeeding bit in the word to the low order bit. If the comparison is unequal, a fault has been detected.

For each simulation pass the machine fault number table (MFNT) (Figure 3) which cross-references each bit position in the host machine word with the fault to which it corresponds, is maintained. Once a comparison is unequal, the table can be entered directly by bit position and the represented fault can be determined.

When a fault is detected, the detection gate records the identification number of the fault detected, the good output vector, the output vector for the fault just detected and the time at which the detection occurred. The input vector, at the time of detection, may be optionally recorded. The important thing to note is that since TEGAS2 simulates on the basis of very accurate propagation delay times, the time of detection has significance. By using this additional information, it is possible to gain increased fault resolution. An example of this is when two faults A and B result in identical output vectors for all input vectors applied. Without any additional information, these two faults cannot be distinguished. However, if the malfunction caused by fault A appears before the malfunction caused by B, then they can be distinguished based on time of detection.

The detection gate performs several other duties. For example, in mode 2 a signal may be flagged indeterminate. In this case, the detection gate checks to see that both the fault induced value and the fault free value are determinate before a fault is declared detected. In all modes of operation, a detection gate may have a clock line as one of its inputs. This clock line or strobe line, may be used to synchronize the process of determining if a fault is detected. In this manner, systems of testers which can examine for faults only at certain time intervals can be accurately simulated.

FAULT CONTROL

When dealing with logic nets of any size at all, such as 500 elements and up, there are thousands of faults to be considered. If such a magnitude of faults is to be simulated efficiently, a good deal of attention must be paid to the overall fault control. The overall control should be such that it will handle an almost unlimited number of faults and be as efficient as possible.

The first step in the process of fault simulation is the specification of the faults to be inserted. As was stated earlier, certain classes of faults may be generated automatically and others specified manually. In either case, the faults are placed sequentially on an external storage device. After all faults have been enumerated, an end-of-file mark is placed on the file and it is rewound. This is accomplished with a control card. The number of faults that can be specified is limited only by the external storage space available.

The maximum number of faults that can be simulated during a single simulation pass is dependent on the number of bits in the host machine word, unless MNFP is used. As mentioned earlier, one bit is always used for the good machine and the others are used to represent fault values. Through the use of a control card, the user may specify the number of bits to be used, up to the maximum allowable.

Let N be the number of faults to be simulated in parallel. The basic steps in fault simulation would then be as follows:

(1) Enumerate all faults to be simulated.
(2) Store on an external device all data necessary to initialize a simulation pass.
(3) Read sequentially N faults from the external fault file and set up the appropriate fault tables.
(4) Negate the appropriate pointers based on the faults tables.
(5) Pass control to the appropriate mode of simulation as determined by the user.
(6) If all faults have been simulated—stop.
(7) If there are more faults to be simulated, restore data necessary to initialize simulation and go to 3.

What has not been explicitly stated, up to this point, is that all input vectors or input stimuli are applied to a group of faults before going to the next group of faults. This will be called the column method. With zero delay or sometimes unit-delay simulation, fault control is not usually done in this manner. In these cases, a single input vector is applied to all groups of faults before going to the next input vector. This will be referred to as the row method. Between applying input vectors in the row method, all faults are examined to determine which ones have been detected and these are discarded. The faults remaining can then be regrouped so that fewer faults need be simulated with the next input vector.

On the surface, the row method control seems more efficient than the column method. However, there are several things to be considered. First of all, when the row method is used with sequential logic, the state information for every fault must be saved at the end of applying each input vector. This requires a great deal of bit manipulation and storage space. The amount of state information that must be stored is dependent on the type of simulation used. If, as with most zero delay simulators, the circuit is leveled and feedback loops are isolated and broken, only the values of feedback loops and flip-flop states need to be stored. With a simulator such as TEGAS2, the circuit is dynamically leveled and feedback loops are never detected and broken, therefore, every signal must be stored. This is one of the reasons that the row method is not considered to be as practical with a time based simulator such as TEGAS2.

A second consideration is the fact that with TEGAS2, all input stimuli can be placed at appropriate places in a time queue before simulation begins. Once simulation begins, it is one continuous process until all stimuli have been applied. Because of this, a large number of input stimuli can be processed very rapidly and efficiently.

The ability to place all input stimuli in a time queue would not be possible if a time based simulator were not used. With a zero delay, or even a unit delay simulator, input stimuli cannot be specified to occur at a particular time in reference to the activity of the rest of the circuit. Therefore, one input vector is applied and the entire circuit must be simulated until it has been determined to be stable. Then the next input vector can be applied, etc. In this manner, there is a certain amount of activity partitioning between input vectors, which lends itself to the row method.

The third factor to consider is that if a fault is no longer simulated after it is once detected, a certain amount of fault isolation information is lost. If the column method is used, the cost of retaining a fault until the desired fault isolation is obtained is considerably less than with the row method.

EXAMPLES

To demonstrate the fault simulation capabilities of TEGAS2, as presented in this paper, consider the network in Figure 8. This network is a particular gate level representation of a J-K master slave flip-flop. The nominal propagational delay time of each of the NAND gates is four (4) time units and the delay of

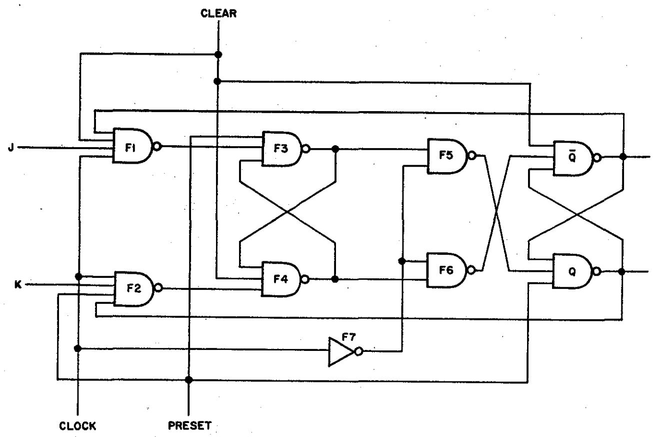


Figure 8—JK master slave flip-flop example

the NOT gate is two (2) time units. The minimum delay of the NAND gates is three (3) time units and the maximum is five (5) time units. For the NOT gate, the minimum and maximum is one (1) and three (3) units, respectively.

The two valued assignable nominal delay mode of simulation is the fastest mode, but it performs no race and hazard analysis. In this mode of simulation, all signals are initially set to zero. Suppose that for the network in Figure 8, the signals J, K, CLEAR, and PRESET are set to 1. Now let the signal CLOCK continuously go up and down with an up time of five and a down time of fifteen. The outputs, Q and $\bar{Q}$ will oscillate because they are never driven to a known state. If the same input conditions are used in the three valued mode of simulation, the outputs will remain constant at X (unknown).

To demonstrate the power of the race and hazard mode of simulation, assume that the inputs J and K change values while the CLOCK is high and that the clock goes to zero two time units after the inputs change. Under these conditions, internal races will be created and a potential error flag will be set for both of the outputs Q and $\bar{Q}$.

Performing the extended fault collapse on the network in Figure 8 resulted in a total of forty faults (Table I) that must be inserted. (A simple gate collapse would result in fifty-two faults to be inserted.)

Table II gives a set of inputs that were applied to the network in all three modes of fault simulation. The table gives all primary input signal changes. In the following analysis of fault detection, the signals Q and $\bar{Q}$ are the only test points for the purpose of observing faults. In the first mode of simulation, two valued assignable nominal delay, thirty-three of the faults were detected. The seven faults not detected were 15, 16, 25, 27, 29, 31, and 33. In the three valued mode of simulation, there were thirty faults declared to be detected. Seven of the faults not detected were the same as in the first mode of simulation. The three other faults not detected were 1, 20, and 23. The reason these faults were not detected, in the three valued mode of simulation, is that they prevented the network from being driven to a known state. In the race and hazard analysis mode of simulation, twenty-six faults were declared to be detected. Out of the fourteen faults not detected, ten are the same as those not detected in the three valued mode. The other four faults are 5, 7, 8 and 11. Faults 7 and 8 were never detected because they never reached a stable state different from a stable state of the good machine's value. Many

TABLE I—Collapsed Set of Faults for Network in Figure 8

Fault No.	Gate	Signal	Fault Type
1		CLEAR	SA1
2		CLEAR	SA0
3		CLOCK	SA1
4		CLOCK	SA0
5		PRESET	SA1
6		PRESET	SA0
7	Q	QB	SA1
8	Q	PRESET	SA1
9		Q	SA0
10	F3	F4	SA1
11	F3	PRESET	SA1
12		F3	SA0
13		F7	SA1
14		F7	SA0
15	F2	Q	SA1
16	F2	PRESET	SA1
17	F2	CLOCK	SA1
18		F2	SA0
19	QB	Q	SA1
20	QB	CLEAR	SA1
21		QB	SA0
22	F4	F3	SA1
23	F4	CLEAR	SA1
24		F4	SA0
25	F1	QB	SA1
26	F1	CLOCK	SA1
27	F1	CLEAR	SA1
28		F1	SA0
29		K	SA1
30		K	SA0
31		J	SA1
32		J	SA0
33	F6	F7	SA1
34	F6	F4	SA1
35		F6	SA1
36		F6	SA0
37	F5	F7	SA1
38	F5	F3	SA1
39		F5	SA1
40		F5	SA0

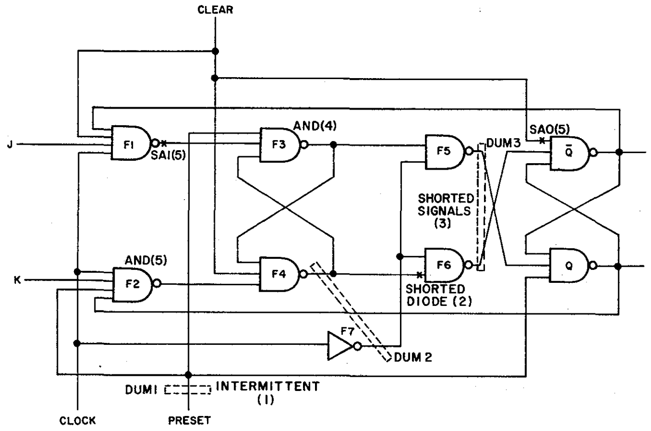


Figure 9—Complex faults

times faults 7 and 8 caused the output signals Q and Q̄ to be in a state of transition while the good machine value was stable. However, this is not sufficient for detection. Faults 5 and 11 caused potential errors and were therefore never declared to be absolutely detected.

Table III gives time vs. faults detected for each of the three modes of simulation. Note that some of the faults are detected at different times between the two valued and three valued modes of simulation. This is due to the fact that some signals were not driven to known states in the three valued mode of simulation until a later time. Faults were also detected at different times in the race and hazard analysis mode since minimum and maximum delay times were used.

Now the insertion of complex faults will be demonstrated. The flip-flop network is marked in Figure 9 with five complex faults. Three of the complex faults require dummy elements. The first fault is an intermittent SA0 on the PRESET signal. The dummy element DUM1 is used to insert this fault. The second fault is an input diode shorted on the connection of signal F4 to gate F6. To insert this fault, the signals F4 and F7 are passed through the dummy element DUM2. The element DUM3 is used to model a signal short between signals F5 and F6. A fourth complex fault is the case where element F3 operates an AND gate instead of a NAND gate. The fifth complex fault is a multiple fault. This multiple fault consists of a SA0 on the input connection of signal CLEAR to gate Q̄, a SA1 on signal F1, and gate F2 operating as an AND gate instead of a NAND gate.

TABLE II—Input Signal Changes

Signal	Value Changed to	Time of Change
J	1	0
K	0	0
CLOCK	0	0
CLEAR	0	0
PRESET	1	0
CLEAR	1	30
CLOCK	1	70
CLOCK	0	110
J	0	131
K	1	131
CLOCK	1	131
CLOCK	0	134
PRESET	0	160

TABLE III—Time vs. Fault Detection

	FAULTS		
Time	Mode-1	Mode-2	Mode-3
4	9, 21	21	
5			21
8	39		
10	1, 6, 20, 38, 40		
12	3, 12, 14		
16	24, 28	6, 24, 28, 38, 40	
18			
20	23		6, 24, 28, 38, 40
46	22, 26	22, 26	
50			22, 26
80	19	19	
83			19
86	13, 37	13, 37	
90			13, 37
120	2, 4, 32	2, 3, 4, 9, 12, 14 32, 39	
123			2, 3, 4, 9, 12, 14 39, 32
124	18, 34, 36	18, 34, 36	
126	10	10	
128			18, 34, 36
130			10
141	7	7	
147	30, 35	30, 35	
150	17	17	
151			17, 30, 37
167	5, 8	5, 8	
179	11	11	

The same input signal changes as given in Table II up through time 110 were applied to the network with these faults present. The times of detection for these faults in mode 1 are:

Time	Fault No.
10	1
12	4
16	3
120	2
120	5

Hence, these faults were simulated simultaneously and detected by the given input sequence for this mode of simulation.

SUMMARY

The TEGAS2 system is capable of simulating faults at three levels. The most accurate level performs race and hazard analysis with minimum and maximum delay times. The fault insertion methods developed for TEGAS2 are capable of modelling not only the traditional set of single occurring stuck-at logical one and stuck-at logical zero faults, but, also a wide range of complex faults such as intermittents, shorted signals, and shorted diodes. In addition, any multiple occurrence of the above faults can be modeled. The specifications of these faults can be done by the user or an extended collapsed set of single occurring stuck-at faults can be generated automatically. Due to accurate time based simulation for faults, it is possible to extract accurate time based fault diagnosis information. Finally, with the introduction of the MNFP technique, a new dimension has been added to digital fault simulation.

REFERFNCES

1 E G ULRICH
Time-sequenced logical simulation based on circuit delay and selective tracing of active network paths
Proceedings ACM 20th National Conference 1965

2 S A SZYGENDA D ROUSE E THOMPSON
A model and implementation of a universal time delay simulator for large digital nets
AFIPS Proceedings SJCC May 1970

3 M A BREUER
Functional partitioning and simulation of digital circuits
IEEE Transactions on Computers Vol C-19 pp 1038–1046 Nov 1970

4 S G CHAPPELL S S YAU
Simulation of large asynchronous logic circuits using an ambiguous gate model
AFIPS Proceedings FJCC November 1971

5 R B WALFORD
The LAMP system
Proceedings of the Lehigh Workshop on Fault Detection and Diagnostics in Digital Circuits and Systems
December 1971

6 R M McCLURE
Fault simulation of digital logic utilizing a small host machine
Proceedings of the 9th ACM-IEEE Design Automation Workshop June 1972

7 E G MANNING H Y CHANG
A comparison of fault simulation methods for digital systems
Digest of the First Annual IEEE Computer Conference 1967

8 S A SZYGENDA
A simulator for digital design verification and diagnosis
Proceedings of the 1971 Lehigh Workshop on Reliability and Maintainability December 1971

9 S A SZYGENDA
TEGAS2—Anatomy of a general purpose test generation and simulation system for digital logic
Proceedings of the 9th ACM-IEEE Deisgn Automation Workshop June 1972

10 D B ARMSTRONG
A deductive method for simulating faults in logic circuits
IEEE Transactions on Computers May 1972

A program for the analysis and design of general dynamic mechanical systems

by D. A. CALAHAN and N. ORLANDEA

The University of Michigan
Ann Arbor, Michigan

INTRODUCTION

The physical laws that govern motion of individual components of mechanical assemblages are well-known. Thus, on the face of it, the concept of a general computer-aided-design program for mechanical system design appears straightforward. However, both the equation formulation and the numerial solution of these equations pose challenging problems for dynamic systems: the former when three-dimensional effects are important, and the latter when the equations become "stiff"[1] or when different types of analyses are to be performed.

In this paper, a three-dimensional mechanical dynamic analysis and design program is described. This program will perform dynamic analysis of nonlinear systems; it will also perform linearized vibrational and modal analysis and automatic iterative design around any solution point in the nonlinear dynamic analysis.

FORMULATION

The equations of motion of a three-dimensional mechanical system can be written in the following form. Free body equations:

$$F_j=\frac{d}{dt}\left(\frac{\partial E}{\partial u_j}\right)-\frac{\partial E}{\partial q_j}-Q_j+\sum_{i=1}^{m}\frac{\partial \phi_i}{\partial q_j}\lambda_i=0 \quad j=1,2,3 \tag{1}$$

$$\left.\begin{aligned} F_j&=\frac{dp_j}{dt}-\frac{\partial E}{\partial q_j}-Q_j+\sum_{i=1}^{m}\frac{\partial \phi_i}{\partial q_j}\lambda_i=0 && (2)\\ F_{j+3}&=p_j-\frac{\partial E}{\partial u_j}=0 && (3)\end{aligned}\right\} j=4,5,6$$

$$F_{j+6}=u_j-q_j=0, \qquad j=1,2,\ldots 6 \tag{4}$$

Constraint (connection) equations:

$$\phi_i=0, \qquad i=1,2,\ldots m \tag{5}$$

where

E is the kinetic energy of the system
q_j are generalized coordinates (three rotational and three translational),
u_j are the coordinate velocities,
λ_i are Lagrange multipliers, representing reaction forces in joints,
p_j are generalized angular momentums,
Q_j are generalized forces,
ϕ_i are constraint functions representing different types of connections at joints (see Figure 2).

Representing all subscripted variables in vector form (e.g., $\underline{u}=[u_1\,u_2\ldots u_6]\mathrm{I}$), these equations become

$$\underline{F}(\underline{\dot{u}},\underline{u},\underline{q},\underline{p},\underline{\lambda};t)=0 \tag{6}$$

$$\underline{\phi}(\underline{q})=\underline{0} \tag{7}$$

By referencing the free body equations of (1-3) to the joints, we can view the above as a "nodal" type of formulation.

NUMERICAL SOLUTION

Static and transient analysis

To avoid the numerical instability associated with widely separated time constants, most general-purpose dynamic analysis programs employ implicit integration techniques. The corrector equation corresponding to (6) has the form

$$\left(\frac{K_0}{T}\frac{\partial F}{d\underline{\dot{u}}}+\frac{\partial F}{\partial \underline{u}}\right)\Delta\underline{u}+\left(\frac{\partial F}{d\underline{q}}\right)\Delta\underline{q}$$

$$+\left(\frac{K_0}{T}\frac{\partial F}{\partial \underline{\dot{p}}}+\frac{\partial F}{\partial \underline{p}}\right)\Delta\underline{p}+\left(\frac{\partial F}{\partial \underline{\lambda}}\right)\Delta\underline{\lambda}=-\underline{F} \tag{8}$$

$$(\partial\phi/\partial\underline{q})\,\Delta\underline{q}=-\underline{\phi} \tag{9}$$

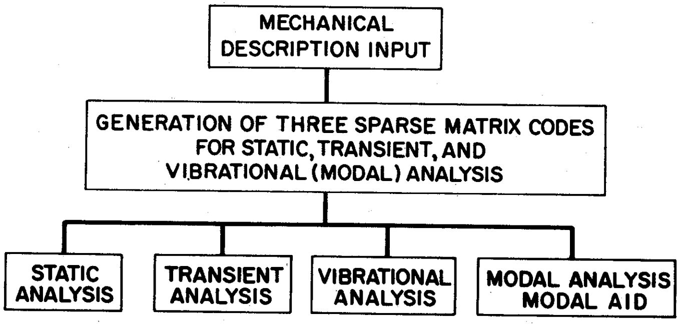

MECHANICAL DESCRIPTION INPUT
GENERATION OF THREE SPARSE MATRIX CODES FOR STATIC, TRANSIENT, AND VIBRATIONAL (MODAL) ANALYSIS
STATIC ANALYSIS
TRANSIENT ANALYSIS
VIBRATIONAL ANALYSIS
MODAL ANALYSIS MODAL AID

Figure 1—Outline of program capabilities

where

T is the integration step size,
K_0 is a constant of integration.

The matrix of partial derivatives in (7-8) is solved repetitively using explicit machine code. The Gear formula is used for integration.[2]

Vibrational and modal analysis

Substitution of s for K_0/T in (8) can be viewed as resulting in the linearized system equations

$$\left[s\left(\frac{\partial F}{\partial \dot{\underline{u}}}\right)^n + \left(\frac{\partial F}{\partial \underline{u}}\right)^n\right]\delta u + \left(\frac{\partial F}{\partial \underline{q}}\right)^n \delta \underline{q}$$

$$+\left(s\frac{\partial F}{\partial \dot{\underline{p}}} + \frac{\partial F}{\partial \underline{p}}\right)\delta \underline{p} + \left(\frac{\partial F}{\partial \underline{\lambda}}\right)^n \delta \underline{\lambda} = \underline{I}(s) \qquad (10)$$

$$(\partial \phi / \partial \underline{q})\delta \underline{q} = \underline{0} \qquad (11)$$

where

$(\ )^n$ represents evaluation at the nth time step; this includes the static equilibrium case ($n=0$),
$\delta\_\_$ represents a small variation around the nth time step,
$I(s)$ is a force or torque source vector.

The evaluation of the vibrational response now proceeds by setting $s=j\omega=j2\pi f$, and sweeping f over the frequency range of interest. This repeated evaluation is similar in spirit to the repeated solution of (8-9) at every corrector iteration. However, now an interpreter[4] is used for solution of the complex-valued simultaneous equations.

Modal analysis (i.e., determination of the natural frequencies) is relatively expensive if all modes must be found. However, the dominant mode can usually be found (from a "good" initial guess) in 5-7 evaluations of the system determinant using Muller's method.[5] This determinant is readily found from the interpreter, which performs an LU factorization to find the vibrational response.

Solution efficiency

For each corrector iteration involved in (8-9), the minimum set of variables that must be determined are those required to update the Jacobian and right hand side vector-$\underline{F}$. The constraint equations represented by $\underline{\phi}=\underline{0}$ can in general relate any q_j variables in a nonlinear manner; also, from (1) and (2), the λ's appear in the $\partial \underline{F}/\partial \underline{q}$ term of the Jacobian. Therefore, it seems convenient to solve for *all* the arguments of $\underline{F}$ of (6). We do not, then, attempt to reduce the number of equations to a "minimum set" (such as the number of degrees of freedom); since most variables must be updated anyway, we find no purpose in identifying such a minimum set for the purpose of transient analysis.

In contrast, for vibrational and modal analysis, a significant savings *could* be achieved by reducing the equations to the number of degrees of freedom of the system. We do not exploit this at present, since a single transient analysis easily dominates other types of analysis in cost.

AUTOMATIC DESIGN

Unlike electrical circuit design, it is not common to design mechanical systems to precisely match a frequency specification. It is far more common to adjust only the dominant mode to achieve an acceptable dynamic response.

One of the most direct approaches to automatic iterative adjustment of the natural modes is to apply Newton iteration to the problem of solving $\Delta(s)=0$

TYPE OF JOINTS	SYMBOL	NO. OF EQUATIONS OF CONSTRAINTS	EXAMPLE OF APPLICATION
SPHERICAL		3	SUSPENSION OF CARS
UNIVERSAL		4	TRANSMISSION FOR CARS
CYLINDRICAL		4	MACHINE TOOLS
TRANSLATIONAL		5	MACHINE TOOLS
REVOLUTE		5	BEARINGS
SCREW		5	SCREWS

Figure 2—Constraint library

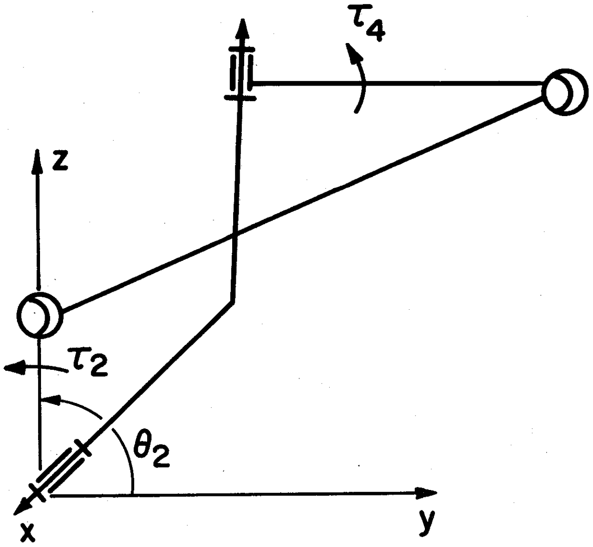


Figure 3—Example mechanism

where Δ is the system determinant associated with (4). In particular, if s_i is a desired natural mode, and ξ_j is a parameter, then we solve iteratively

$$\sum_j \left[\frac{\partial \Delta(s_i)}{\partial \xi_j}\right] \Delta\xi_j = -\Delta(s_i)$$

or

$$\sum_i \left[\left(\frac{\partial \Delta(s_i)}{\partial \xi_j}\right) \Big/ \Delta(s_i)\right] \Delta\xi_j = -1$$

Here, the term in brackets can be identified as a transfer function of the linearized system.

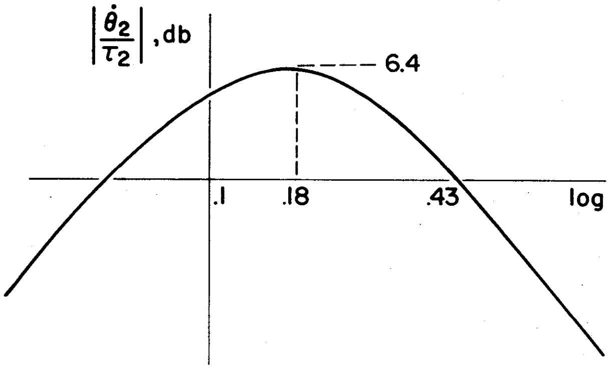


Figure 5—Vibrational response

PROGRAM DESCRIPTION

The general features of the program are outlined in Figure 1. In general, the program is intended to permit analysis of assemblages of links described by their masses and three inertial moments compatibly connected by any of the joints of Figure 2. Other types of mechanical elements (gears, cams, springs, dashpots) will be added shortly. Most of these fit neatly into the nodal formulation, effecting only the constraint equations given in (2).

EXAMPLE

The system shown in Figure 3 was simulated over a duration of 42.5 seconds of physical time. It was as-

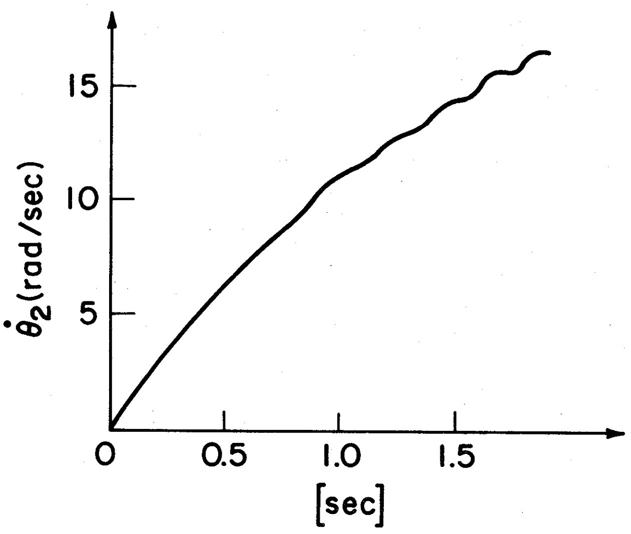


Figure 4—Transient response

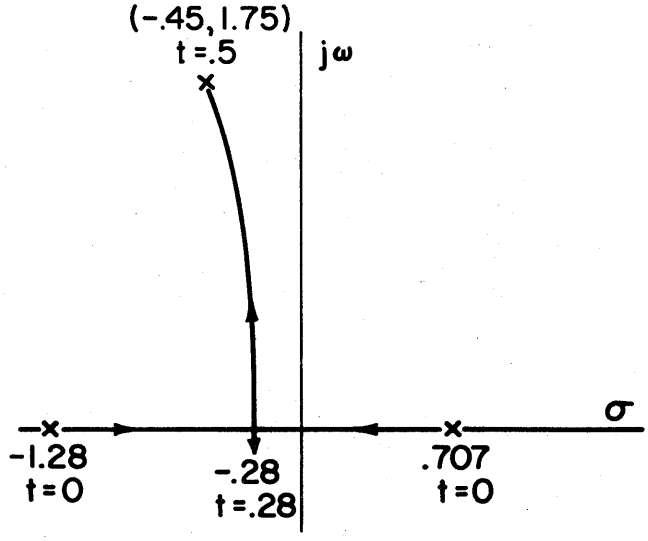


Figure 6—Locus of natural modes

sumed that a motor with a linear torque vs. speed characteristic, τ_2 vs. ω_2, drove the system against a constant load torque, τ_4. Figure 4 shows the transient response; a vibrational response is shown in Figure 5 around the static equilibrium point. Figure 6 shows the motion of the natural frequencies as the transient response develops; it may be noted that as the natural modes develop a larger imaginary component, an oscillation of increasing frequency appears in the transient response.

SUMMARY

The nodal formulation of (1-5) offers a number of programming and numerical solution features.

(1) No topological preprocessing is necessary to establish a set of independent variables; equations can be developed directly from the connection data, component-by-component.

(2) Having a large number of solution variables assists in modeling common physical phenomena; for example, frictional effects in joints are routinely modeled and impact is easily handled.

(3) Sensitivity necessary for man-machine and iterative design are easily determined due to the explicit appearance of common parameters (e.g., masses, inertial terms, link dimensions) in the nodal formulation.

(4) The use of force and torque equations permits easy capability with current methods of continuum mechanics for internal stress analysis.

It must be mentioned that the transient solution of three-dimensional mechanical systems poses some interesting numerical problems not present in the related fields of circuit and structural analysis. First, the equations are highly nonlinear requiring evaluations of tensor products at each corrector iteration; also, associated matrices are of irregular block structure. Second, the natural modes are not infrequently in the right half plane, representing a falling or a locking motion (see Figure 3). The integration of such (locally) unstable equations is not a well-understood process and can be expected to yield some numerical difficulties. Among these appear to be a high degree of oscillation in the reaction forces (λ_i's), preventing any effective error control to be exerted on these variables.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the interest and support of the U. S. Air Force Office of Scientific Research (Grant No. AFOSR-71-2027), and the National Science Foundation (NSF Grant No. GK-31800).

REFERENCES

1 C W GEAR
DIFSUB for solution of ordinary differential equations CACM
Vol 14 No 3 pp 185-190 March 1971

2 N ORLANDEA M A CHACE D A CALAHAN
Sparsity-oriented methods for simulation of mechanical dynamic systems
Proc 1972 Princeton Conf on Information Sciences and Systems March 1972

3 F G GUSTAVSON W M LINIGER R A WILLOUGHBY
Symbolic generation of an optimal count algorithm for sparse systems of linear equations
Sparse Matrix Proceedings (1969) IBM Thomas J Watson Research Center Yorktown Heights New York Sept 1971

4 H LEE
An implementation of gaussian elimination for sparse systems of linear equations
Sparse Matrix Proceedings (1969) IBM Thomas J Watson Research Center Yorktown Heights New York Sept 1971

5 D E MULLER
A method for solving algebraic equations using an automatic computer
Math Tables Aids Computer Vol 10 pp 208-215 1956

6 M A CHACE D A CALAHAN N ORLANDEA D SMITH
Formulation and numerical methods in the computer evaluation of mechanical dynamic systems
Proc Third World Congress for the Theory of Machines and Mechanisms Kupari Yugoslavia pp 61-99 Sept 13-20 1971

7 R C DIX T J LEHMAN
Simulation of dynamic machinery
ASME Paper 71-Vibr-111 Proc Toronto ASME Meeting Sept 1971

A wholesale retail concept for computer network management

by DAVID L. GROBSTEIN

Picatinny Arsenal
Dover, New Jersey

and

RONALD P. UHLIG

US Army Materiel Command
Washington, D.C.

THE MANAGEMENT PROBLEM

In the past few years the technical feasibility of computer networks has been demonstrated. An examination of the existing networks, however, indicates that they are generally composed of homogeneous machines or are located essentially in one geographical area. The most notable exception to this is the ARPA Network which is widely distributed geographically and which has a variety of computers. The state-of-the-art now appears to be sufficiently far along to allow serious consideration of computer networks which are not experimental in origin and are not university based.

When a large governmental or industrial organization contemplates the establishment of a computer network, initial excitement' focuses on technical sophistication and capabilities which may be achieved. As the problem is examined more deeply it becomes progressively clearer that the management aspects represent the greater challenge. There are a number of sound reasons for an organization to establish a computer network, but fundamental to these is the intent to reduce over-all computer resources required, by sharing them. The implications of this commitment to share are more far reaching than is immediately obvious when the idea is first put forth.

In both government and industry it is common to find computing facilities established to service the needs of a particular profit center or activity. That is, the computer resources necessary to support a mission organization are placed under its own control, as in Divisions 1, 2, and 4 in Figure 1.

The commitment to share computer resources in a network implies substantial changes in the resource control topology of that organization. This is particularly true for organizations that have existing computing facilities which will be pooled to form the base of the network's resources. The crux of the matter is that sharing implies not only that you will let someone else utilize the unused capacity of your computer; it also implies that you may be told to forgo installing your own machine because there is unused capacity elsewhere in the resource pool. If your mission depends on the availability and suitability of computer services from someone else's machine you suddenly become very interested in the management structure which governs the relationship between your organization and the one that has the computer.

The purpose of this paper is to examine some of the objectives and problems of an organization having existing independent computing centers, when it contemplates moving into a network environment.

COMPUTER NETWORK ADVANTAGES

Computer network advantages can be divided into two categories, operational and management. The following advantages are classified as operational in that they affect the day to day use of facilities in the network:

1. Provide access to large scale computers by users who do not have on-site machines.

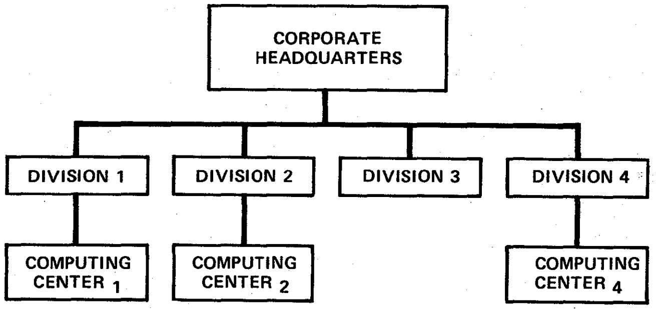


Figure 1—Decentralized computing facilities

2. Provide access to different kinds of computers that are available in the network.
3. Provide access to specialized programs or technology available at particular computing centers.
4. Reduce costs by sharing of proprietary programs without paying for them at multiple sites.
5. Load level among computing centers.
6. Provide back-up capability in case of over-load, temporary, or extended outage.

A very fundamental advantage is the provision of a full range of computing power to users, without having to install a high capacity machine at each site. To achieve this, it is necessary to provide access to the network through time sharing, batch processing and interactive graphics terminals. For each of these to be applied to that portion of a project for which it is best suited, all must have access to a common data base.

Computer users frequently find programs that will be valuable to them but which have been developed for some other machine. Conversion to their own machine can be time consuming and costly even if the programs are written in FORTRAN. Computer networks can offer access to different kinds of machines so that borrowed programs may be run without conversion. If the program will serve without modification it need not be borrowed at all but can be used through the network at whichever installation has developed it. Thus a network environment can be used to encourage specialized technology at each computing center so that implementation and maintenance costs need not be repeated at every user's site.

Computer networks can provide better service to users by allowing load leveling among the centers in the network so that no single machine becomes so overloaded that response and turn-around time degrade to unacceptable levels. Furthermore, the availability of like machines provides back up facilities to insure relatively uninterrupted service, at least for high priority work.

From the standpoint of managing computer resources, networks offer several advantages in helping to achieve the goal of the best possible service for the least cost. Among these advantages are:

1. Greater ease and precision in identifying aggregated computing workload requirements, by providing a larger and more stable base from which to make workload projections.
2. Ability to add capacity to the network as a whole, rather than at each individual installation by developing specifications for new main frames based on total network requirements, with less regard for specific geographic location.
3. Computing power can be added in increments which more closely match requirements.

Experience at a number of installations indicates that it is extremely difficult to project computer use on a project by project basis with sufficient accuracy to use the aggregated data as a basis for installation or augmentation of computer facilities. Project estimations vary widely, particularly in scientific and engineering areas. The need for computer support is strongly driven by the week to week exigencies of the project. Because of this variability, larger computing centers can often project their future requirements better from past history and current use trends, than by adding up the requirements of each individual project. In a computing network these trends can be more easily identified, and, since the network as a whole serves a larger customer base than any single installation, the projections can be made more accurately. Simply stated, the law of large numbers applies to the aggregate.

The second and third management advantages listed above are interrelated but not really the same. In a network, adding capacity to any node makes that capacity available to everyone else in the network. It is important to recognize that this applies to specialized kinds of capacity as well as to general purpose computer cycles. Thus when specifications for new hardware are developed, they can include requirements derived from the total network. Finally computer capacity tends to come in fixed size pieces. In the case of computers which can service relatively large and relatively long running computer programs, the pieces are not only large, they are very expensive. When these have to be provided at each installation requiring computer services, there is frequently expensive unused capacity when the equipment is first installed. In a network, added computing power can be more easily matched to overall requirements because the network capacity increments are distributed over a larger base.

WHOLESALE VS RETAIL FUNCTIONS

Now let's examine the services obtained from a computing network.

At most large computing centers, personnel, financial, and facilities resources are devoted to a combination of functions which include acquisition and operation of computing hardware, installation and maintenance of operating systems, language processors, and other general purpose "systems" software, and design and development of applications programs. These functions are integrated by the computing center manager to try to provide the best overall service to his customers. The Director of Computing Services at a location with its own computer center, provides an organizational interface with his local customers which may include the Director of Laboratories, the Director of Research and Engineering, Director of Product Assurance, and other similar functions which require scientific and engineering computer support.

But what structure is required if there is no computing center at a particular location? How does the use of computer network services, instead of organizationally local hardware, affect the computer supported activities? Conversely, in a computer network environment, what is the effect of having no customers at the actual local site of the computing center? What functional structure is required at such a lonely center and what services should it offer?

In considering the answer to these and other questions involved in the establishment of a computer network it is useful to distinguish wholesale from retail computing services. At its most fundamental level the wholesale computing function might be defined as the production of usable computer cycles. In order to achieve this it is necessary to have not only computer hardware, but also the operating systems software, language processors, etc., which are needed to make the hardware cycles accessible and usable. The wholesaler produces his services in bulk. The production of wholesale computer cycles may be likened to the production of coal, oil, or natural gas. Each of these products can be used in support of a wide variety of applications from the production of electricity to heating homes to broiling steaks on the back yard grill. The specific application is not the primary concern of the wholesaler. His concern is to produce bulk quantities of his product at the lowest possible cost. The Wholesale Computing Facility (WCF) like the oil producer, has to offer a well-defined, stable product, in a sufficient number of grades (classes of service) to satisfy his end users. To achieve this he also must have a marketing function which interacts with his retailers in order to maximize the value of the products he offers. The marketing function includes technical representatives in the form of software and hardware consultants which can explain to the retailer how to derive the maximum value from the services offered and how to solve technical problems which arise.

TABLE I—Resources and Services Offered by A Typical Wholesale Computing Facility (WCF) ;

RESOURCES	SERVICES
Computers	Batch processing access
System Software	Interactive terminal access
General Purpose Application Software	Realtime access
	Data File storage
Systems Programmers	Data Base Management
Operators	Contract programming
Communications Equipment	Consulting Services
	Systems Software
	Hardware Interfaces
	Communications
	Documentation & Manuals
	Marketing/Marketing Support

Table I is a non-exhaustive list of the resources needed and services offered by a typical WCF.

Unlike the Wholesale Computing Facility which strives for efficient and effective production of general purpose computing power, the Retail Computing Facility (RCF) has the function of efficiently and effectively delivering service directly to the user. The user's concern is with mission accomplishment. He has a project to complete, and the computer provides an analytical tool. He is not directly concerned with efficiency of computer operation; he is concerned with maximizing the value of computer services to his project. In this respect fast turn-around time and specialized applications programs which ease his burden of communicating with the computer may be more important than obtaining the largest number of computer cycles per dollar. The retailer's function is to provide an interface between the WCF and the user. His primary concern is to cater to the special needs, the taste and style of his customers. He must provide a wide variety of services which tailor the available computing power to each specialized need.

To do this it is vital that the retailer understand and relate to his user's needs and capabilities. For the sophisticated user he may have to provide interactive terminal access and a variety of high level languages with which the user can develop his own specialized applications programs. For others he must offer analyst and programmer services to develop computer

TABLE II—Resources Needed By and Services Offered By a Typical Retail Computing Facility (RCF)

RESOURCES	SERVICES
Wholesale/Retail Agreements	Usable Computer Time
Access to computers (terminals)	Special Purpose Applications Programming
Personnel	General Purpose Applications Programming
General Purpose Applications Programs	Software Consultant Services
Marketing Support	Applications and Debugging Consultation
	User Training
	Administrative Services Arrangement for Terminals Users Guides, Manuals, Key Punching, Password Assignments Marketing

applications to the customer's specifications. His primary orientation must be toward supporting his user's missions.

The Retail Computing Facility also represents its users to the Wholesale Computing Facilities. In doing so, it helps the wholesaler to determine the kind of products which must be offered. The retailer may need to buy batch processing, interactive time sharing, and computer graphics services. He may need access to several different brands of computers, in order to process applications programs which his users have developed or acquired from others. He acquires commitments for these services from wholesalers through wholesale/retail agreements. Table II indicates resources needed and services provided by RCFs.

UTILITY OF THE WHOLESALE RETAIL DISTINCTION

The notion of separate wholesale and retail computing facilities is useful for several reasons, particularly when a large company or government agency is attempting to integrate independent decentralized computing centers into a network. In the pre-network environment both the wholesale and retail facilities tend to be contained in the same organization and have responsibility for servicing only that organization. In a network environment it is important to identify the Wholesale Computing Facility in order to understand that it will be serving other organizations as well, and therefore must take a non-parochial point of view. The importance of this viewpoint is indicated by the fact that wholesale/retail agreements are regarded by the retailer as a resource. For the retailer to depend on them, the agreements must be binding, and the retailer must be assured that he will receive the same treatment when he is accessing a computer remotely through the network as he would if he were geographically and organizationally a part of the wholesaler's installation. After all, to achieve the benefits of sharing computer resources which a network offers, it is necessary to tell some organizations that they cannot have their own computers. Thus it is clear that binding agreements, as surrogates for local computer centers, are fundamental to successful network implementation.

Another reason to distinguish between wholesale and retail facilities is to make it clear that you cannot serve users merely by placing bare terminals where they can be reached. Examination of the retail functions indicate that they include a large number of the user oriented services offered by existing computing centers. It is important to recognize that the decision to use only terminal access to the network at some locations, does not result in saving all the resources that would be required to set up an independent computing center at those locations. Quite the contrary, if computing services are needed, it is a management obligation to provide the required resources for a successful Retail Computing Facility.

A third reason for identifying the two functions is that in discussing organization and funding, lines of responsibility and control are clearer to portray. This third reason implies that the wholesale/retail distinction is useful in understanding and planning for network organization, whether or not the distinction becomes visible in the implemented organization as separate

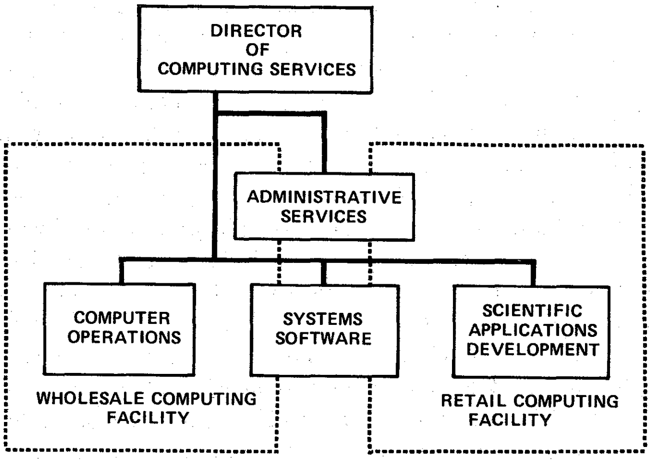

DIRECTOR OF COMPUTING SERVICES
ADMINISTRATIVE SERVICES
COMPUTER OPERATIONS
SYSTEMS SOFTWARE
SCIENTIFIC APPLICATIONS DEVELOPMENT
WHOLESALE COMPUTING FACILITY
RETAIL COMPUTING FACILITY

Figure 2—Wholesale and retail computing facilities identified within a "typical" computing center

segments. The wholesale and retail portions of a "typical" computing center are indicated in Figure 2.

APPLICATION OF THE WHOLESALE/RETAIL MANAGEMENT CONCEPT

The concepts discussed to this point were developed in a search for answers to some very real problems currently facing the authors' organization. We want to make it clear that these theories and ideas are not official policy of our organization; rather they are possible solutions to some of these problems. In discussing the approach described above with colleagues throughout our organization, we discovered that it is useful to consider possible applications of these ideas in concrete rather than abstract terms. Our colleagues needed to know where they fit into the plan in order to understand it. Furthermore, mapping a general plan onto the structure of a specific organization is a prerequisite to acceptance.

THE AUTHORS' ORGANIZATION

The authors are in scientific and engineering data processing management positions with the US Army Materiel Command (AMC), a major command of the US Army employing approximately 130,000 civilians, and 13,000 military at the time this paper was written. AMC has the mission of carrying out research, development, testing, procurement, supply and maintenance of the hardware in the Army's inventory. The scope of this mission is staggering. Some of the major organizational elements comprising the Army Materiel Command include "Commodity Commands" with responsibility for research, development, procurement and supply for specific groups of commodities (hardware), depots for maintenance and supply, and independent laboratories for exploratory research.

Because of the nature of its mission, AMC might be likened to a large corporation with many divisions. For example, one of the "Commodity Commands"—Tank Automotive Command in Detroit, Michigan—carries out work similar to that carried out by a major automobile manufacturer in the United States. Another "Commodity Command"—Electronics Command—carries out work similar to that carried out by a major electronics corporation. In a sense each of these "Commodity Commands" operates as a small corporation within the larger parent corporation. Each Commodity Command has laboratory facilities for carrying out research in its areas of commodity responsibility. In addition, independent laboratories carry out basic and exploratory research. It may be helpful in the discussion which follows to draw a comparison between industrial situations and the Army Materiel Command. The Commanding General of AMC occupies a position similar to that of the President of a large diversified corporation. The Commanding Generals of each of the Commodity Commands and independent laboratories might be compared to Senior Group Vice Presidents in this large corporation, while the Commanding Officers of various research activities within Commodity Commands carry out functions similar to those carried out by Vice Presidents responsible for particular mission areas within a corporation.

As in many large corporations, AMC has a number of different types of computers in geographically dispersed locations to provide computer support under many different Commanding Officers.

Locations having major computing resources which are candidates for sharing, and locations requiring scientific and engineering computer support are shown in Figure 3. The resources which are candidates for sharing include 8 IBM 360 series computers (1 model 30, 1 model 40, 2 model 44s, 1 model 50, 3 model 65s), three Control Data Corporation 6000 series computers (1 CDC 6500, and 2 CDC 6600s), seven Univac 1100 series computers (6 Univac 1108s, 1 Univac 1106), one Burroughs 5500 computer, two EMR 6135 computers, and two additional major computers not yet selected. These 21 computers are located and operated at 17 different locations among those shown in Figure 3. It is not clear that every one of the locations requiring

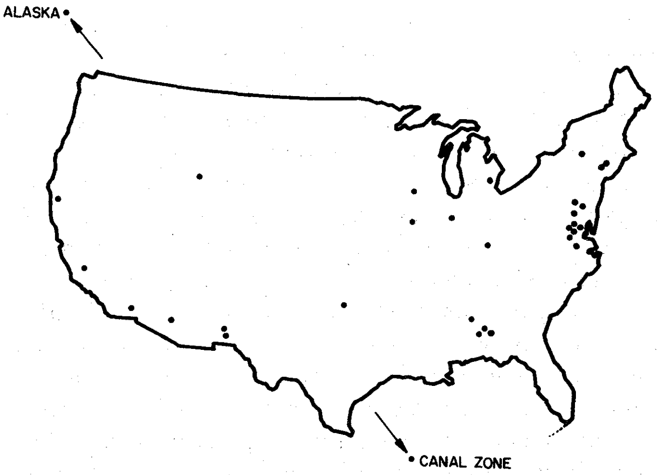


Figure 3—AMC locations which have scientific computers or require scientific computing services

services should ultimately receive them through a computing network. The main purpose of this illustration is to show the magnitude of the problem.

In exploring the existing situation it came as somewhat of a surprise to discover that we already have most of the management problems of computer networks, despite the fact that not all of the seventeen computer sites and thirty-one users sites are interconnected. Computer support agreements now exist between many different activities within Army Materiel Command, although not all of these provide for service via terminals.

DECENTRALIZED MANAGEMENT OF THE NETWORK NODES

The wholesale/retail organizing rationale discussed previously was developed as a vehicle for better understanding our present management structure, and as an aid in identifying a viable structure for pooling computer resources across major organizational boundaries.

A number of proposals to centralize operational management of all of these computers were considered and discarded. The computing centers which would form the network exist today, and most have been operational for a number of years. They are well managed and running smoothly and we would like to keep it that way. Furthermore, the association of these centers with the activities which they serve has been mutually beneficial. ("Activity" is used here to refer to an organizational entity having a defined mission and distinct geographic location.) The centers receive resource support from the activities and in turn provide for the specialized needs of the research and development functions which they serve. Sharing of these specialized technologies and services is a desirable objective of forming the network. For these reasons, the authors believe decentralized computer management would be necessary for a successful network.

To make our commitment to decentralized computer management viable, we needed to face squarely the issue that each existing computer is used and controlled by a local Commanding Officer to accomplish the assigned research and development mission of his activity. But network pooling of computer resources implies that some activities use the network in lieu of installing their own computer. For this approach to succeed, availability of time in the computer pool has to be guaranteed to approximately the same degree as would derive from local hardware. The offered guarantee in a network environment would be an agreement between the activity with the computer and the activity requiring computer support. To make the network succeed, corporate (or AMC) headquarters would have to set policies insuring that agreements have sufficient force to guarantee the using organization the resources specified.

In the following paragraphs we will discuss how these agreements might be used in the Army Materiel Command type of environment. If we replace the words "Commanding Officer" with the words "Vice President" it seems clear that the same concepts apply to industry as well as to the military situation.

If agreements are to become sufficiently binding so that they can be considered a resource it would be necessary to expand the basic mission of the Commanding Officer who "owns" the computer. The only way to make the computer into a command-wide (or corporate) resource would be to assign the Commanding Officer and his Director of Computing Services the additional mission of providing computer support to all organizations authorized to make agreements with him, and to identify the resources under his control which would be given the task of providing computer services to "outside" users. These resources would now become a Wholesale Computing Facility serving both local and outside organizations.

FUNCTIONS OF THE RETAILER

In a large corporation with many divisions each division would require a "retailer" of computer services to perform the applications oriented data processing. Those divisions which operate computers would operate them as wholesale functions to provide computer service to all divisions within the corporation. Substituting the words "Commodity Command, Major Subordinate Command, or Laboratory" for the word "Division" the same principle could apply to the Army Materiel Command. Although a local commander would give up some control over "his" computer, in that he would guarantee some capacity to outside users, he would gain access to capacity on every other computer within the command, to support him in accomplishing his primary mission.

Retail Computing Facility (RCF) describes that part of the organization responsible for assuring that computer services are available to the customers and users to accomplish the primary mission of the local activity. Every scientist and engineer within an activity, e.g., laboratory, would look to his local RCF to provide the type of service required. The RCF would turn to wholesalers throughout the entire corporation. This would give the retailer the flexibility to fit to the job an available computer rather than having to force fit the job on

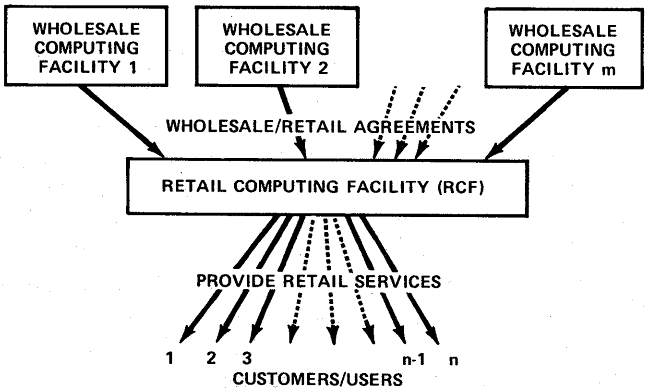

WHOLESALE COMPUTING FACILITY 1
WHOLESALE COMPUTING FACILITY 2
WHOLESALE COMPUTING FACILITY m
WHOLESALE/RETAIL AGREEMENTS
RETAIL COMPUTING FACILITY (RCF)
PROVIDE RETAIL SERVICES
1
2
3
n-1
n
CUSTOMERS/USERS

Figure 4—RCF uses wholesale service agreements with several WCFs to provide retail services to customers

to the local computer. These relationships are shown in Figure 4.

In order to obtain the resources and provide the services listed in Table II a considerable amount of homework would have to be done by the retailer. The retailer would estimate the types and amounts of services required by his various users and arrange agreements with wholesalers to obtain these services. It must be recognized that this is a difficult job and in many instances cannot be done with great accuracy. The retailer would act as a middleman between customer/users and Wholesale Computer Facilities within the network.

The retailer would be responsible for negotiating two different types of agreements. He would have to negotiate long term commitments with various wholesalers by guaranteeing to these wholesalers a certain minimum dollar amount; in return the wholesalers would guarantee to the retailers a certain minimum amount of computer time.

The other type of agreement which retailers could negotiate with wholesalers would be for time as required. This would take the form of a commitment to spend dollars at a particular wholesale facility when the demand occurred and if time were available from that wholesaler. The retailer could then run jobs at that WCF on a "first come, first served" basis, or according to whatever queue discipline was agreed upon in advance. The range of agreements would be from "hard scheduled computer runs" to "time as available."

It is imperative that a user not have to go through lengthy negotiation each time he requests computer service from a retailer. Submitting a job through the local retailer to any computer in the corporate network should be at least as simple as the current procedures for submitting a job to a local computer at a user's home installation.

FUNCTIONS OF THE WHOLESALER

Figure 5 graphically depictes the Wholesale Computer Facility relationship to retailers. The WCF at installation *m* would provide resources and services listed in Table 1 through the network to retailers at various activities throughout the corporation. Normally, the WCF at installation *m* would still provide most of the service to the retailer at installation *m*; however, that retailer would not have any formally privileged position over other retailers located elsewhere in the network. The primary functions of the wholesaler would be to operate the computers and to provide the associated services which have been negotiated by various retailers. The wholesaler might well have services which were duplicated elsewhere in the network; however, he might also have some which were unique to his facility. It would be essential that every retailer in the network be made aware of the services offered by each WCF, and it would be the responsibility of the wholesaler to ensure that all of his capabilities were made known.

In addition to operating the computer or computers at his home installation, the WCS might also be responsible for providing services to retailers through contracts placed with facilities external to the corporation. For example, a propriety software package not available from any computer in the corporate pool, but required by one or more retailers, might be available from some other computer which could be accessed by the corporate net. A contract to access those services could then be placed through one of the wholesalers.

HEADQUARTERS FUNCTIONS IN MANAGING THE CORPORATE NETWORK

Corporate Headquarters interaction with decentralized wholesale and retail computing facilities can be

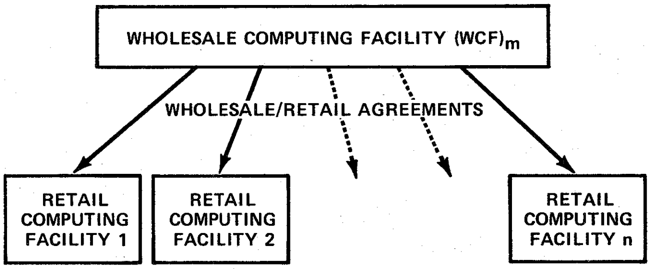

WHOLESALE COMPUTING FACILITY (WCF)m
WHOLESALE/RETAIL AGREEMENTS
RETAIL COMPUTING FACILITY 1
RETAIL COMPUTING FACILITY 2
RETAIL COMPUTING FACILITY n

Figure 5—WCFs provide service to multiple RCFs

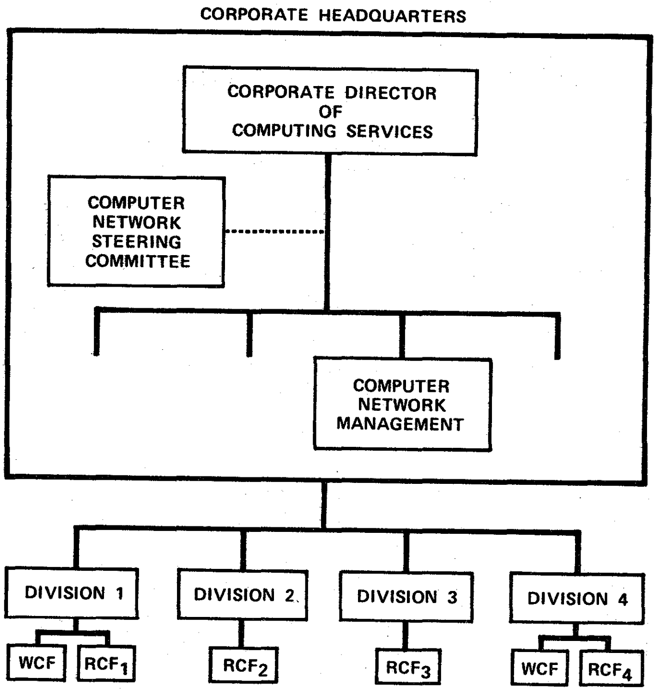

CORPORATE HEADQUARTERS
CORPORATE DIRECTOR OF COMPUTING SERVICES
COMPUTER NETWORK STEERING COMMITTEE
COMPUTER NETWORK MANAGEMENT
DIVISION 1
DIVISION 2
DIVISION 3
DIVISION 4
WCF
RCF_1
RCF_2
RCF_3
WCF
RCF_4

Figure 6—Corporate headquarters organization with decentralized WCFs and RCFs

provided through the establishment of two groups, Computer Network Management (CNM), and the Computer Network Steering Committee (CNSC). Both CNM and CNSC should report to the Corporate Director of Computing Services as shown in Figure 6.

Overall, the headquarters is responsible for insuring that computer support requirements of scientists and engineers throughout the corporation are effectively met, and that they are provided in an efficient manner. The first responsibility is to insure that proper computer support is available. The second responsibility is to insure that the minimum amount of dollars are expended in providing that support.

Computer Network Management (CNM) has basic headquarters staff responsibility to insure that the network is well coordinated and well run. It should:

1. Recommend policy and procedures for regulation and operation of the network.
2. Resolve network problems not covered by corporate procedures.
3. Negotiate facilities management agreements with the appropriate corporation divisions to operate Wholesale Computing Facilities (see Figure 6).
4. Work with WCFs, RCFs and the Computer Network Steering Committee to develop long range plans concerning network facilities.
5. Serve as a network-wide information center on facilities, services, rates, and procedures.

CNM need not be directly involved in the day to day operations of the network. Wholesale/retail agreements should be negotiated between WCFs and RCFs without requiring headquarters approval, so long as these agreements are consistent with overall corporate policy. Obviously, agreements not meeting this requirement would require CNM involvement. However, headquarters should function, insofar as possible, on a management by exception basis.

A Computer Network Steering Committee (CNSC) should be established to suggest policy for consideration by the corporation. Members of the CNSC should be drawn from the corporation's operating divisions which have responsibility for decentralized management of the wholesale and retail computing facilities. The Computer Network Steering Committee can promote input to the Corporate Headquarters of useful comments and ideas on network policy and operation.

Under the general structure some specific functions in which Computer Network Management would be involved can be discussed further.

Policies set by Computer Network Management should govern the content of agreements between wholesalers and retailers. The following is a list of some of the items which would have to be covered in such agreements:

1. The length of time for which an agreement should run would have to be spelled out in each case.
2. The wholesaler would have to guarantee a specific amount of service to the retailer in return for a guarantee of a minimum number of dollars from the retailer.
3. The kinds and levels of service to be provided would have to be spelled out in detail.

Another major area in which Computer Network Management could be involved is in setting rates and in rationing services during periods of congestion. Policies should be established which would promote as effective and efficient support as possible during congested periods, without starving any single customer. Also, the total amount of computer time which each wholesaler can commit should be regulated to prevent over commitment of the network.

Computer Network Management should set up some form of "currency" to be used when resources become congested. The amount of "currency" in the network would be regulated by Computer Network Management with advice from the Computer Network Steering Committee. This "currency" based rationing scheme should be put into effect ahead of time, rather than waiting until resources become so congested that it has to be created under emergency conditions. It is probable that separate rations should be established for different classes of service, such as interactive terminal service, fast turn around batch processing, overnight turn around, etc.

Under this organizational concept the corporate headquarters would have to assume a greater responsibility for projecting requirements and procuring new hardware and software to meet those requirements throughout the corporation. Some requirements would arise which would have to be met immediately. There would not always be sufficient time to purchase new hardware or software. In such cases computer network management could arrange for external service contracts to be let through one or more wholesalers. CNM would have the responsibility for identifying peak workloads anticipated for the entire network on the basis of feedback information received from wholesalers and retailers. When overall network services become congested, an open ended external service contract might be placed to handle the excess. This provides time for a corporate decision to be made as to whether or not additional computing capacity should be added to the network.

The last major responsibility of Computer Network Management would be to aggregate requirements being received from wholesalers and retailers and to use these to project when new hardware and software should be procured for the S&E network community. The primary responsibility for justifying this new hardware would rest with CNM, drawing on all corporate resources for support and coordination. Computer Network Management with guidance from the Computer Network Steering Committee, would also be responsible for determining where new hardware should be placed in order to run the network in the most effective and efficient manner.

Computer Network Management would fulfill its mission of insuring computer support to corporate scientists and engineers by negotiating facility management agreements with specific divisions of the corporation to establish and operate Wholesale Computer Facilities. These WCFs would offer the specified kinds and levels of service to Retail Computer Facilities via the network. RCFs would tailor and add to the services to meet requirements of local customers.

SUMMARY

The notions of wholesale and retail computer facilities are particularly useful in examining the problems which must be faced when entering a computer network environment. The concept helps to clarify the functions which must be performed within a network of shared computer resources, and the management commitment which must be made if the objectives and advantages of such sharing are to be realized. Mapping of the wholesale/retail functions onto the corporate organization which is forming the network can be valuable in identifying to members of that organization what their roles would be in the network environment. Such clarification is a prerequisite to securing the commitment necessary to make a network successful.

Decisions as to whether or not operational management of the computer centers should be decentralized will vary with circumstances, but if efficient, well managed decentralized computing facilities exist, they should be retained. In any case, a central computer network management function is needed to set policy and to take an overall corporate viewpoint. It should be remembered, however, that the primary purpose of a scientific and engineering computing network is to provide services to research and development projects at field activities. As such, the goal should be to contribute to the optimization of the costs and time involved in the research and development cycle, rather than to optimize the production of computer cycles. The establishment of a network steering committee which includes representatives from field activities can help to insure the achievement of this goal and to increase confidence in the network among the field personnel which it is to serve.

Finally it is important to realize that a corporation begins to enter the network environment, from the management standpoint, as soon as some of its major activities begin to share computer resources, whether or not it involves any computer to computer communications facilities. Recognition of this point and a careful examination of corporate objectives and goals in computer sharing should lead to the establishment of a computer network management function, so that the corporation can manage itself into an orderly network environment rather than drifting into a chaotic one.

ACKNOWLEDGMENT

The authors would like to gratefully acknowledge extensive discussions and interaction with a group of people whose ideas and hard work contributed sub-

stantially to the content of this paper: Mr. Einar Stefferud, Einar Stefferud & Associates; and the following members of organizations within the US Army Materiel Command: Richard Butler, Harry Diamond Labs; John Cianflone, US Army Materiel Command Headquarters; James Collins, Missile Command; Tom Dames, Electronics Command; Edward Goldstein, Test and Evaluation Command; Dr. James Hurt, Weapons Command; Paul Lascala, Aviation Systems Command; Sam P. McCutchen, Mobility Equipment R&D Center; James Pascale, Watervliet Arsenal; Michael Romanelli, Aberdeen Research & Development Center; George Sumrall, Electronics Command.

REFERENCES

1 E STEFFERUD
A wholesale/retail structure for the AMC computer network
Unpublished Discussion Paper Number ES&A/AMC/CNC DP-1 February 3 1972

2 J J PETERSON S A VEIT
Survey of computer networks
Mitre Corporation
MTP-357 September 1971

3 F P BROOKS J K FERRELL T M GALLIE
Organizational, financial, and political aspects of a three university computing center
Proceedings of the IFIP Congress 1968 E49-52

4 M S DAVIS
Economics—point of view of designer and operator
Proceedings of Interdisciplinary Conference on Multiple Access Computer Networks
University of Texas and Mitre Corporation 1970

5 J J HOOTMAN
The computer network as a marketplace
Datamation Vol 18 No 4 April 1972

6 C MOSMANN E STEFFERUD
Campus computing management
Datamation Vol 17 No 5 March 1971

7 E STEFFERUD
Computer management
College and University Business September 1970

8 L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
AFIPS Conference Proceedings May 1970

9 F E HEART et al
The interface message processor for the ARPA computer network
AFIPS Conference Proceedings May 1970

10 C S CARR S D CROCKER V G CERF
HOST-HOST communication protocol in the ARPA network
AFIPS Conference Proceedings May 1970

11 E STEFFERUD
Management's role in networking
Datamation Vol 18 No 4 April 1972

12 E STEFFERUD
The environment of computer operating system scheduling: Toward an understanding
Journal of the Association for Education Data Systems March 1968

13 BLUE RIBBON DEFENSE PANEL
Report to the President and the Secretary of Defense on the Department of Defense Appendix I: Staff report on automatic data processing
July 1970

14 S D CROCKER et al
Function-oriented protocols for the ARPA computer network
AFIPS Conference Proceedings
May 1970

A functioning computer network for higher education in North Carolina

by LELAND H. WILLIAMS

Triangle Universities Computation Center
Research Triangle Park, North Carolina

INTRODUCTION

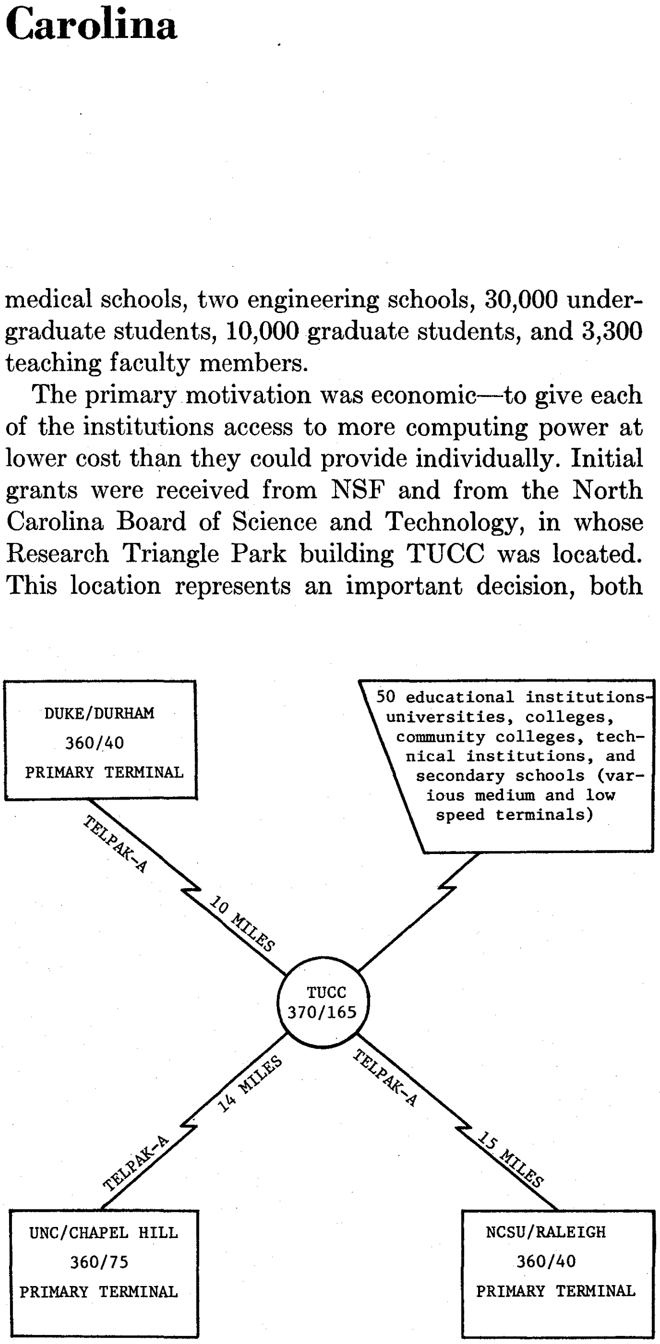
Currently there is a great deal of talk concerning computer networks. There is so much such talk that the solid achievements in the area sometimes tend to be overlooked. It should be clearly understood then, that this paper deals primarily with achievements. Only the last section, which is clearly labeled, deals with plans for the future.

Adopting terminology from Peterson and Veit,[1] TUCC is essentially a centralized, homogeneous network comprising a central service node (IBM 370/165), three primary job source nodes (IBM 360/75, IBM 360/40, IBM 360/40) and 23 secondary job source nodes (leased line Data 100s, UCC 1200s, IBM 1130s, IBM 2780s, and leased and dial line IBM 2770s) and about 125 tertiary job source nodes (64 dial or leased lines for Teletype 33 ASRs, IBM 1050s, IBM 2741s, UCC 1035s, etc.) See Figures 1 and 2. All source node computers in the network are homogeneous with the central service node and, although they provide local computational service in addition to teleprocessing service, none currently provides (non-local) network computational service. However, the technology for providing network computational service at the primary source nodes is immediately available and some cautious plans for using this technology are indicated in the last section of this paper.

BACKGROUND

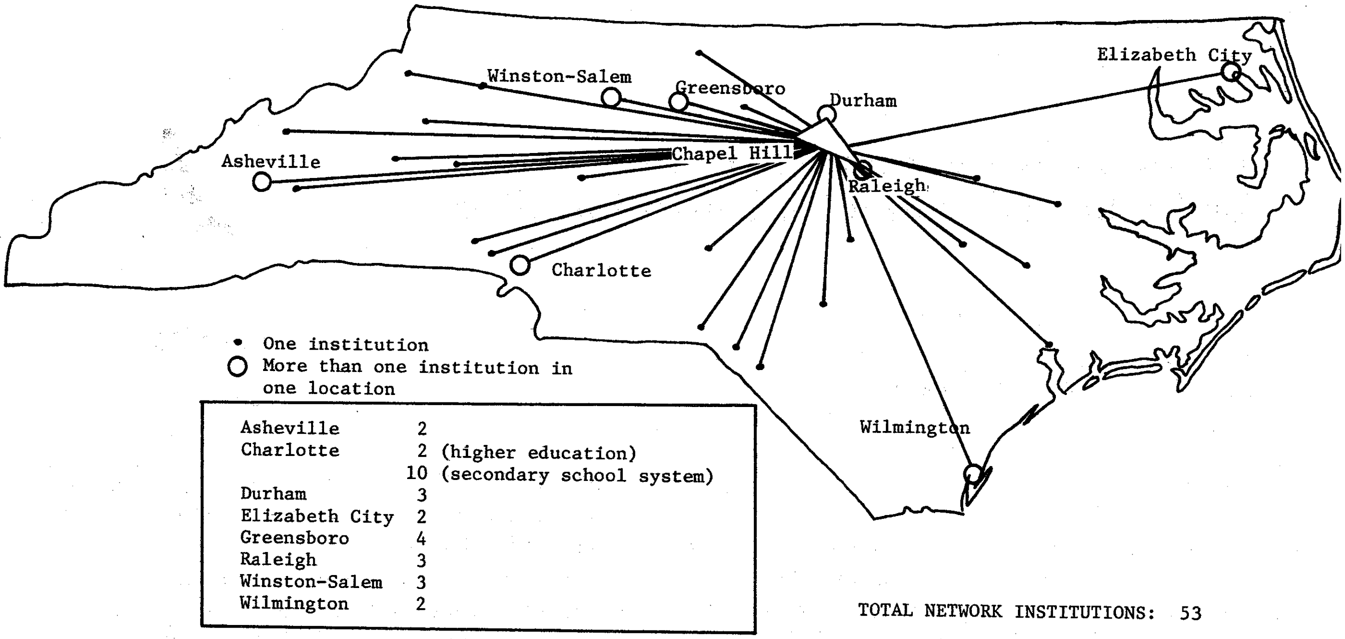
The Triangle Universities Computation Center was established in 1965 as a non-profit corporation by three major universities in North Carolina—Duke University at Durham, The University of North Carolina at Chapel Hill, and North Carolina State University at Raleigh. Duke is a privately endowed institution and the other two are state supported. Among them are two medical schools, two engineering schools, 30,000 undergraduate students, 10,000 graduate students, and 3,300 teaching faculty members.

The primary motivation was economic—to give each of the institutions access to more computing power at lower cost than they could provide individually. Initial grants were received from NSF and from the North Carolina Board of Science and Technology, in whose Research Triangle Park building TUCC was located. This location represents an important decision, both



NOTE: IN ADDITION TO THE PRIMARY TERMINAL INSTALLATION AT DUKE, UNC, AND NCSU, EACH CAMPUS HAS AN ARRAY OF MEDIUM AND LOW-SPEED TERMINALS DIRECTLY CONNECTED TO TUCC.

Figure 1—The TUCC network



Asheville	2	
Charlotte	2	(higher education)
	10	(secondary school system)
Durham	3	
Elizabeth City	2	
Greensboro	4	
Raleigh	3	
Winston-Salem	3	
Wilmington	2	

TOTAL NETWORK INSTITUTIONS: 53

Figure 2—Network of institutions served by TUCC/NCECS

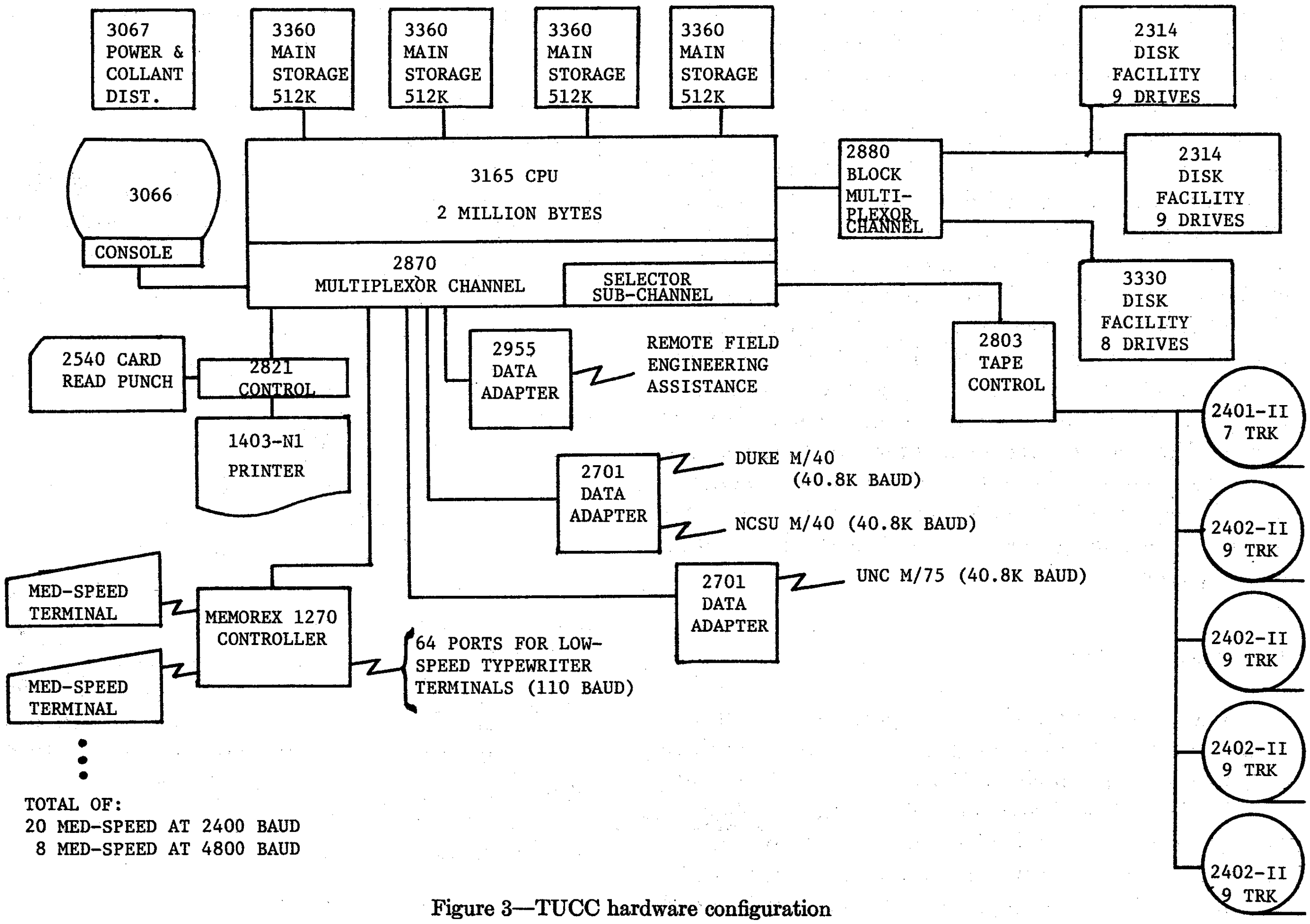


Figure 3—TUCC hardware configuration

because of its geographic and political neturality with respect to all three campuses and because of the value of the Research Triangle Park environment.

The Research Triangle Park is one of the nation's most successful research parks. In a wooded tract of 5,200 acres located in the small geographic triangle formed by the three universities, the Park in 1972 has 8,500 employees, a payroll of $100 million and an investment in buildings of $140 million. The Park contains 40 buildings that house the research and development facilities of 19 separate national and international corporations and government agencies and other institutions.

TUCC pioneered massively shared computing; hence there were many technological, political, and protocol problems to overcome. Successive stages toward solution of these problems have been reported by Brooks, Ferrell, and Gallie;[2] by Freeman and Pearson;[3] and by Davis.[4] This paper will focus on present success.

PRESENT STATUS

TUCC supports educational, research, and (to a lesser, but growing extent) administrative computing requirements at the three universities, and also at 50 smaller institutions in the state and several research laboratories by means of multi-speed communications and computer terminal facilities. TUCC operates a 2-megabyte, telecommunications-oriented IBM 370/165 using OS/360-MVT/HASP and supporting a wide variety of terminals (see Figure 3). For high speed communications, there is a 360/75 at Chapel Hill and there are 360/40s at North Carolina State and Duke. The three campus computer centers are truly and completely autonomous. They view TUCC simply as a pipeline through which they get massive additional computing power to service their users.

The present budget of the center is about $1.5 million. The Model 165 became operational on September 1, 1971, replacing a saturated 360/75 which was running a peak load of 4200 jobs/day. The life of the Model 75 could have been extended somewhat by the replacement of 2 megabytes of IBM slow core with an equal amount of Ampex slow core. This would have increased the throughput by about 25 percent for a net cost increase of about 8 percent.

TUCC's minimum version of the Model 165 costs only about 8 percent more than the Model 75 and it is expected to do twice as much computing. So far it has processed 6100 jobs/day without saturation. This included about 3100 autobatch jobs, 2550 other batch jobs, and 450 interactive sessions. Of the autobatch jobs, 94 percent were processed with less than 30 minutes delay (probably 90 percent with less than 15 minutes delay), and 100 percent with less than 3 hours delay. Of all jobs, 77 percent were processed with less than 30 minutes delay, and 99 percent with less than 5 hours delay. At the present time about 8000 different individual users are being served directly. The growth of TUCC capability and user needs to this point is illustrated in Figure 4.

Services to the TUCC user community include both remote job entry (RJE) and interactive processing. Included in the interactive services are programming systems employing the BASIC, PL/1, and APL languages. Also TSO is running in experimental mode. Available through RJE is a large array of compilers including FORTRAN IV, PL/1, COBOL, ALGOL, PL/C, WATFIV and WATBOL. These language facilities coupled with an extensive library of application programs provide the TUCC user community with a dynamic information processing system supporting a wide variety of academic computing activities.

ADVANTAGES

The financial advantage deserves further comment. As a part of the planning process leading to installation

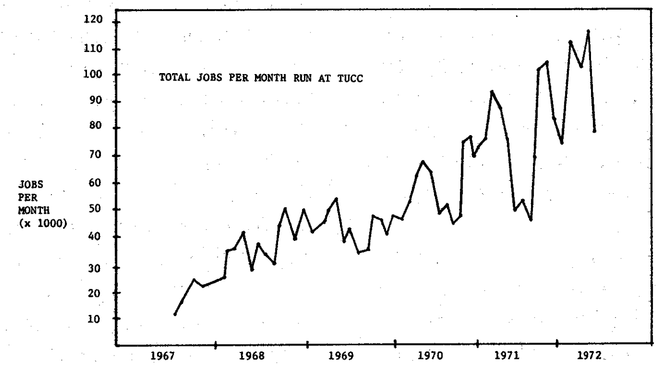


Figure 4—TUCC jobs per month, 1967-1972

of the Model 165, one of the universities concluded that it would cost them about $19,000 per month more in hardware and personnel costs to provide all their computing services on campus than it would cost to continue participation in TUCC. This would represent a 40 percent increase over their present expense for terminal machine, communications, and their share of TUCC expense.

There are other significant advantages. First, there is the sharing of a wide variety of application programs. Once a program is developed at one institution, it can be used anywhere in the network with no difficulty. For proprietary programs, usually only one fee need be paid. A sophisticated TUCC documentation system sustains this activity. Second, there has been a significant impact on the ability of the universities to attract faculty members who need large scale computing for their research and teaching and several TUCC staff members including the author have adjunct appointments with the university computer science departments.

A third advantage has been the ability to provide very highly competent systems programmers (and management) for the center. In general, these personnel could not have been attracted to work in the environment of the individual institutions because of salary requirements and because of system sophistication considerations.

NORTH CAROLINA EDUCATIONAL COMPUTING SERVICE

The North Carolina Board of Higher Education has established an organization known as the North Carolina Educational Computing Service (NCECS). This is the successor of the North Carolina Computer Orientation Project[5] which began in 1966. NCECS participates in TUCC and provides computer services to public and private educational institutions in North Carolina other than the three founding universities. Presently 40 public and private universities, junior colleges, and technical institutes plus one high school system are served in this way. NCECS is located with TUCC in the North Carolina Board of Science and Technology building in the Research Triangle Park. This, of course, facilitates communication between TUCC and NCECS whose statewide users depend upon the TUCC telecommunication system.

NCECS serves as a statewide campus computation center for their users, providing technical assistance, information services, etc. In addition, grant support from NSF has made possible a number of curriculum development activities. NCECS publishes a catalog of available instructional materials; they provide curriculum development services; they offer workshops to promote effective computer use; they visit campuses, stimulating faculty to introduce computing into courses in a variety of disciplines. Many of these programs have stimulated interest in computing from institutions and departments where there was no interest at all. One major university chemistry department, for example, ordered its first terminal in order to use an NCECS infrared spectral information program in its courses.

The software for NCECS systems is derived from a number of sources in addition to sharing in the community wide program development described above. Some of it is developed by NCECS staff to meet a specific and known need; some is developed by individual institutions and contributed to the common cause; some of it is found elsewhere, and adapted to the system. NCECS is interested in sharing curriculum oriented software in as broad a way as possible.

Serving small schools in this way is both a proper service for TUCC to perform and is also to its own political advantage. The state-supported founding universities, UNC and NCSU, can show the legislature how they are serving much broader educational goals with their computing dollars.

ORGANIZATION

TUCC is successful not only because of its technical capabilities, but also because of the careful attention given to administrative protection of the interests of the three founding universities and of the NCECS schools. The mechanism for this protection can, perhaps, best be seen in terms of the wholesaler-retailer concept.[6] TUCC is a wholesaler of computing service; this service consists essentially of computing cycles, an effective operating system, programming languages, some application packages, a documentation service, and management. The TUCC wholesale service specifically does *not* include typical user services—debugging, contract programming, etc. Nor does it include user level billing nor curriculum development. Rather these services are provided for their constituents by the Campus Computation Centers and NCECS, which are the retailers for the TUCC network. See Figure 5.

The wholesaler-retailer concept can also be seen in the financial and service relationships. Each biennium, the founding universities negotiate with each other and with TUCC to establish a minimum financial commitment from each, to the net budgeted TUCC costs. Then, on an annual basis the founding universities and TUCC negotiate to establish the TUCC machine configuration, each university's computing resource share, and the cost to each university. This negotiation, of course,

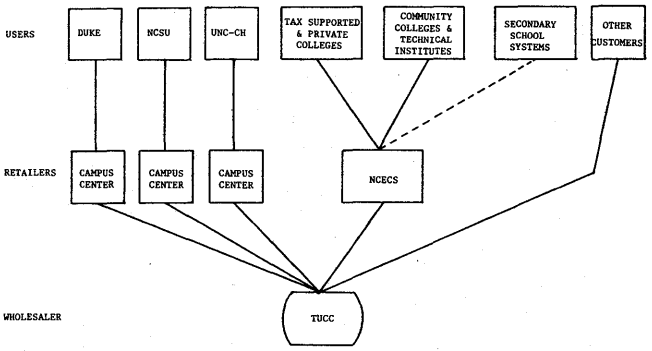


Figure 5—TUCC wholesaler-retailer structure

includes adoption of an operating budget. Computing resource shares are stated as percentages of the total resource each day. These have always been equal for the three founding universities, but this is not necessary. Presently each founding university is allocated 25 percent, the remaining 25 percent being available for NCECS, TUCC systems development, and other users. This resource allocation is administered by a scheduling algorithm which insures that each group of users has access to its daily share of TUCC computing resources. The algorithm provides an effective trade-off for each category between computing time and turn-around time; that is, at any given time the group with the least use that day will have job selection preference.

The scheduling algorithm also allows each founding university and NCECS to define and administer quite flexible, independent priority schemes. Thus the algorithm effectively defines independent sub-machines for the retailers, providing them with the same kind of assurance that they can take care of their users' needs as would be the case with totally independent facilities. In addition, the founding university retailers have a bonus because the algorithm defaults unused resources from other categories, including themselves, to one or more of them according to demand. This is particularly advantageous when their peak demands do not coincide. This flexibility of resource use is a major advantage which accrues to the retailers in a network like TUCC.

The recent installation of the old TUCC Model 75 at UNC deserves some comment at this point because it represents a good example of the TUCC organization in action. UNC has renewed a biennial agreement, with its partners, calling essentially for continued equal sharing in the use of and payment for TUCC computing resources. Such equality is possible in our network precisely because each campus is free to supplement as required at home. Further more, the UNC Model 75 is a very modest version of the prior TUCC Model 75. It has 256K of fast core and one megabyte of slow core where TUCC had one and two megabtyes respectively. Rental accruals and state government purchase plans combined to make the stripped Model 75 cost UNC less than their previous Model 50. It provides only a 20 percent throughput improvement over the displaced Model 50. The UNC Model 75 has become the biggest computer terminal in the world!

There are several structural devices which serve to protect the interests of both the wholesaler and the retailers. At the policy making level this protection is afforded by a Board of Directors appointed by the Chancellors of the three founding universities. Typically each university allocates its representatives to include (1) its business interests, (2) its computer science instructional interests, and (3) its other computer user interests. The University Computation Center Directors sit with the Board whether or not they are members as do the Director of NCECS and the President of TUCC. A good example of the policy level function of this Board is their determination, based on TUCC management recommendations, of computing service rates for NCECS and other TUCC users.

At the operational level there are two important groups, both normally meeting each month. The Campus Computation Center Directors' meeting includes the indicated people plus the Director of NCECS and the President, the Systems Manager, and the Assistant to the Director of TUCC. The Systems Programmers' meeting includes representatives of the three universities, NCECS and TUCC. In addition, of course, each of the universities has the usual campus computing committees.

PROSPECTS

TUCC continues to provide cost-effective general computing service for its users. Some improvements which can be foreseen include:

1. A wider variety of interactive services to be made available through TSO.
2. An increased service both for instructional and administrative computing for the other institutions of higher education in North Carolina.
3. Additional economies for some of the three founding universities through increasing TUCC support of their administrative data precessing requirements.
4. Development of the network into a multiple service node network by means of the symmetric HASP-to-HASP software developed at TUCC.
5. Provision (using HASP) for medium speed terminals to function as message concentrators for

low speed terminals, thus minimizing communication costs.
6. Use of line multiplexors to reduce communication costs.
7. Extension of terminal service to a wider variety of data rates.

Administrative data processing

Some further comment can be made on item 3. TUCC has for some time been handling the full range of administrative data processing for two NCECS universities and is beginning to do so for other NCECS schools. The primary reason that this application lags behind instructional applications in the NCECS schools is simply that grant support, which stimulated development of the instructional applications, has been absent for administrative applications. However, the success of the two pioneers has already began to spread among the others.

With the three larger universities there is a greater reluctance to shift their administrative data processing to TUCC, although Duke has already accomplished this for their student record processing. One problem which must be overcome to complete this evolution and allow these universities to spend administrative computing dollars on the more economic TUCC machine is the administrator's reluctance to give up a machine on which he can exercise direct priority pressure. The present thinking is that this will be accomplished by extending the sub-machine concept (job scheduling algorithm) described in the previous section so that each founding university may have both a research-instructional sub-machine and an administrative sub-machine with unused resources defaulting from either one to the other before defaulting to another category. Of course, the TUCC computing resource will probably have to be increased to accommodate this; the annual negotiation among the founders and TUCC provides a natural way to define any such necessary increase.

SUMMARY

Successful massively shared computing has been demonstrated by the Triangle Universities Computation Center and its participating educational institutions in North Carolina. Some insight has been given into the economic, technological, and political factors involved in the success as well as some measures of the size of the operation. The TUCC organizational structure has been interpreted in terms of a wholesale-retail analogy. The importance of this structure and the software division of the central machine into essentially separate sub-machines for each retailer cannot be over-emphasized.

REFERENCES

1 J J PETERSON S A VEIT
Survey of computer networks
MITRE Corporation Report MTP-359 1971

2 F P BROOKS J K FERRELL T M GALLIE
Organizational, financial, and political aspects of a three university computing center
Proceedings of the IFIP Congress 1968 E49-52

3 D N FREEMAN R R PEARSON
Efficiency vs responsiveness in a multiple-service computer facility
Proceedings of the 1968 ACM Annual Conference

4 M S DAVIS
Economics—point of view of designer and operator
Proceedings of Interdisciplinary Conference on Multiple Access Computer Networks
University of Texas and MITRE Corporation 1970

5 L T PARKER T M GALLIE F P BROOKS J K FERRELL
Introducing computing to smaller colleges and universities—a progress report
Comm ACM Vol 12 1969 319-323

6 D L GROBSTEIN R P UHLIG
A wholesale retail concept for computer network management
AFIPS Conference Proceedings Vol 41 1972 FJCC

Multiple evaluators in an extensible programming system*

by BEN WEGBREIT

Harvard University
Cambridge, Massachusetts

INTRODUCTION

As advanced computer applications become more complex, the need for good programming tools becomes more acute. The most difficult programming projects require the best tools. It is our contention that an effective tool for programming should have the following characteristics:

(1) Be a complete *programming system*—a language, plus a comfortable environment for the programmer (including an editor, documentation aids, and the like).
(2) Be *extensible*, in its data, operations, control, and interfaces with the programmer.
(3) Include an *interpreter* for debugging and several *compilers* for various levels of compilation—all fully compatible and freely mixable during execution.
(4) Include a *program verifier* that validates stated input/output relations or finds counter-examples
(5) Include facilities for *program optimization and tuning*—aids for program measurement and a subsystem for automatic high-level optimization by means of source program transformation.

We will assume, not defend, the validity of these contentions here. Defenses of these positions by us and others have appeared in the literature.[1,2,3,4,5] The purpose of this paper is to discuss how these characteristics are to be simultaneously realized and, in particular, how the evaluators, verifier, and optimizer are to fit together. Compiling an extensible language where compiled code is to be freely mixed with interpreted code presents several novel problems and therefore a few unique opportunities for optimization. Similarly, extensibility and multiple evaluators make program automation by means of source level transformation more complex, yet provide additional handles on the automation process.

This paper is divided into five sections. The second section deals with communication between compiled and interpreted code, i.e., the runtime information structures and interfaces. The third section discusses one critical optimization issue in extensible languages—the compilation of unit operations. The fourth section examines the relation between debugging problems, proving the correctness of programs, and use of program properties in compilation. Finally, the fifth section discusses the use of transformation sets as an adjunct to extension sets for application-oriented optimization.

Before treating the substantive issues, a remark on the implementation of the proposed solutions may be in order. Our acquaintance with these problems has arisen from our experience in the design, implementation, and use of the ECL programming system. ECL is an extensible programming system utilizing multiple evaluators; it has been operational on an experimental basis, running on a DEC PDP10, since August 1971. Some of the techniques discussed in this paper are functional, others are being implemented, still others are being designed. As the status of various points is continually changing, explicit discussion of their implementation state in ECL will be omitted.

For concreteness, however, we will use the ECL system and ECL's base language, EL1, as the foundation for discussion. An appendix treats those aspects of EL1 syntax needed for reading the examples in this paper.

* This work was supported in part by the U.S. Air Force, Electronics Systems Division, under Contract F19628-71-C-0173 and by the Advanced Research Projects Agency under Contract F19628-71-C-0174.

MIXING INTERPRETED AND COMPILED CODE

The immediate problem in a multiple evaluator system is mixing code. A program is a set of procedures which call each other; some are interpreted, others compiled by various compilers which optimize to various levels. Calls and non-local gotos are allowed where either side may be either compiled or interpreted. Additionally, it is useful to allow control flow by means of RETFROM—that is the forced return from a specified procedure call (designated by name), with a specified value as if that procedure call had returned normally with the given value (cf. Reference 6).

Within each procedure, normal techniques apply. Interpreted code carries the data type of each entity—for autonomous temporary results as well as parameters and locals. Since the set of data types is open-ended and augmentable during execution, data types are implemented as pointers to (or indices in) the *data type table*. Compiled code can usually dispense with data types so that temporary results need not, in general, carry type information. In either interpreted or compiled procedures, where data types are carried, the type is associated *not* with the object but rather with a descriptor consisting of a type code and a pointer to the object. This results in significant economies whenever objects are generated in the free storage region.

Significant issues arise in communication *between* procedures. The interfaces must:

(1) Allow identification of free variables in one procedure with those of a lower access environment and supply data type information where required.
(2) Handle a special, but important, subcase of #1—non-local gotos out of one procedure into a lower access environment.
(3) Check that the arguments passed to compiled procedure are compatible with the formal parameter types.
(4) Check that the result passed back to a compiled procedure (from a normal return of a called function or via a RETFROM) is compatible with the expected data type.

These communication issues are somewhat complicated by the need to keep the overhead of procedure interface as low as possible for common cases of two compiled procedures linking in desirable (i.e., well-programmed) ways.

The basic technique is to include in the binding (i.e., parameter block) for any new variable its *name* and its *mode* (i.e., its data type) in addition to its value. Names are implemented as pointers to (or indices in) the *symbol table*. (With reasonable restrictions on the number of names and modes, both *name* and *mode* can be packed into a 32-bit word.) Within a compiled procedure, all variables are referenced as a pair ⟨block level number, variable number within that block⟩. Translation from name to such a reference pair is carried out for each bound appearance of a variable during compilation; at run time, access is made using a *display* (cf. Reference 7). However, a free appearance of a variable is represented and identified by symbolic *name*. Connection between the free variable and some bound variable in an enclosing access environment is made during execution, implemented using either *shallow* or *deep* bindings (cf. Reference 8 for an explanation of the issues and a discussion of the trade-offs for LISP). Once identification is made, the mode associated with the bound variable is checked against the *expected* mode of the free variable, if the expected mode is known.

To illustrate the last point, we suppose that in some procedure, P, it is useful to use the free variable BETA with the knowledge that in all correctly functioning programs the relevant bound BETA will always be a character string. To permit partial type checking during compilation, a declaration may be made at the head of the first BEGIN-END block of P.

```
DECL BETA:STRING SHARED BETA;
```

This creates a local variable BETA of mode STRING which *shares* storage (i.e., binding by reference in FORTRAN or PL/I[9]) with the free variable BETA. All subsequent appearances of BETA in P are *bound*, i.e., identified with the local variable named BETA. Since the data type of the local BETA is known, normal compilation can be done for all internal appearances of BETA. The real identity of BETA is fixed during execution by identification with the free BETA of the access environment at the point P is entered. When the identification of bound and free BETA is made, mode checking (e.g., half-word comparison of two type codes) ensures that mode assumptions have not been violated.

In the worst case, parameter bindings entail the same sort of type checking. The arguments passed to a procedure come with associated modes. When a procedure is entered, the *actual argument modes* can be checked against the expected parameter modes and, where appropriate, conversion performed. Then the *names* of the formal parameters are added to the argument block, forming a complete variable binding. Notice that this works in all four cases of caller/callee pairs: compiled/

compiled, compiled/interpreted, interpreted/compiled and interpreted/interpreted. Since type checking is implemented by a simple (usually half-word) comparison, the overhead is small.

However, for the most common cases of compiled/compiled pairs, mode checking is handled by a less flexible but more efficient technique. The mode of the called procedure may be declared in the caller. For example:

```
    DECL G:PROC(INT,STRING;COMPLEX);
```

specifies that G is a procedure-valued variable which takes two arguments, an integer and a character string, and returns a complex number. For each call on G in the range of this declaration, mode checking and insertion of conversion code can be done during compilation, with the knowledge that G is constrained to take on only certain procedure values. To guarantee this constraint, all assignments to (or bindings of) G are type checked. Type checking is made relatively inexpensive by giving G the mode PROC(INT,STRING;COMPLEX)—i.e., there is an entry in the *data type table* for it—and comparing this with the mode of the procedure value being assigned. The single comparison simultaneously verifies the validity of the result mode and both argument modes.

Result types are treated similarly. For each procedure call, a uniform call block is constructed* which includes the name of the procedure being called and the expected mode of the result (e.g., for the above example, the *name* field is G and the *expected-result-mode* field is COMPLEX). This is ignored when compile-time checking of result type is possible and normal return occurs. However, if interpreted code returns to compiled code, or if RETFROM causes a return to a procedure by a non-direct callee, then the expected-result-mode field is checked against the mode of the value returned.

Transfer of control to non-local labels falls out naturally if labels are treated as named entities having constant value. On entry to a BEGIN-END block (in either interpreted or compiled code), a binding is made for each label in that block. The label value is a triple ⟨indicator of whether the block is interpreted or compiled, program address, stack position⟩. A non-local goto label L is executed by identifying the label value referenced by the free use of L, restoring the stack position from the third component of the triple, and either jumping to the program address in compiled code or to the statement executor of the interpreter.

UNIT COMPILATION

In most programs the bulk of the execution time is spent performing the unit operations of the problem domain. In some cases (e.g., scalar calculations on reals), the hardware realizes the unit operations directly. Suppose, however, that this is not the case. Optimizing such programs requires recognizing instances of the unit operations and special treatment—*unit compilation*—to optimize these units properly.

An extensible language makes recognition a tractable problem, since the most natural style of programming is to define data types for the unit entities, and procedures for the unit operations in each problem area. (Operator extension and syntax extension allow the invocation of these procedures by prefix and infix expressions and special statement types.) Hence, the unit operations are reasonably well-modularized. Detecting which procedures in the program are the critical unit operations entails static analysis of the call and loop structure, coupled with counts of call frequency during execution of the program over benchmark data sets.

The critical unit operations generally have one or more of the following characteristics:

(1) They have relatively short execution time; their importance is due to the frequency of call, not the time spent on each call.
(2) Their size is relatively small.
(3) They are terminal nodes of the call structure, or nearly terminal nodes.
(4) They entail a repetition, performing the same action over the lower-level elements which collectively comprise the unit object of the problem level.

Unit compilation is a set of special heuristics for exploiting these characteristics.

Since execution time is relatively small, call/return overhead is a significant fraction. Where the unit operations are terminal, the overhead can be substantially reduced. The arguments are passed from compiled code to a terminal unit operation with no associated modes. (Caller and callee know what is being transmitted.) The arguments can usually be passed directly in the registers. No bindings are made for the formal parameters. (A terminal node of the call structure calls no other; hence, there can be no free uses of these variables.) The result can usually be returned in a register again, with no associated mode information.

Since the unit operations are important far out of proportion to their size, they are subject to optimizing techniques too expensive for normal application. Opti-

* This can be included in the LINK information.[7]

mal ordering of a computation sequence (e.g., to minimize memory references or the number of temporary locations) can, in general,* be assured only by a search over a large number of possible orderings. Further, the use of identities (e.g., $a*b+a*c \rightarrow a*(b+c)$) to minimize the computational cost causes significant increase in the space of possibilities to be considered. The use of arbitrary identities, of course, makes the problem of program equivalence (and, hence, of cost minimization) undecidable. However, an effective procedure for obtaining equivalent computations can be had either by restricting the sort of transformations admitted[11] or by putting a bound on the degree of program expansion acceptable. Either approach results in an effective procedure delivering a very large set of equivalent computations. While computationally intractable if employed over the whole program, a semi-exhaustive search of this set for the one with minimal cost is entirely reasonable to carry out on a small unit operator. Similarly, to take full advantage of multiple hardware function units, it is sometimes necessary to unwind a loop and rewind it with a modified structure—e.g., to perform, on the *i*th iteration of the new loop, certain computation which was formerly performed on the $(i-1)$st, *i*th, and $(i+1)$st iteration. Again, a search is required to find the optimal rewinding.

In general, code generation which tries various combinations of code sequences and chooses among them (by analysis or simulation) can be used in a reasonable time scale if consideration is restricted to the few unit operations where the pay-off is significant. Consider, for example, a procedure which searches through an array of packed k-bit elements counting the number of times a certain (parameter-specified) k-bit configuration occurs. The table can either be searched in array order—all elements in the first word, then all elements in the next, etc.—or in position order—all elements in the first position of a word, all elements in the next position, etc. Which search strategy is optimal depends on k, the hardware for accessing k-bit bytes from memory, the speed of shifting vs. memory access, and the sort of mask and comparison instructions for k-bit bytes. In many situations, the easiest way of choosing the better strategy is to generate code for each and compute the relative execution times as a function of array length.

A separate issue arises from non-obvious unit operations. Suppose analysis shows that procedures F and G are each key operations (i.e., are executed very frequently). It may well be that the appropriate candidates for unit compilation are F, G, and some particular combination of them, e.g., "F;G" or "G(...F(...)...)". That is, if a substantial number of calls on G are preceded by calls of F (in sequence or in an argument position), the new function defined by that composition should be unit compiled. For example, in dealing with complex arithmetic, +, −, *, /, and CONJ are surely unit operations. However, it may be that for some program, "u/v+v*CONJ(v)" is critical. Subjecting this combination to unit compilation saves four of the ten multiplications as well as a number of memory frequencies.

ASSUMPTIONS AND ASSERTIONS

If an optimizing compiler is to generate really good code, it must be supplied the same sort of additional information that would be given to or deduced by a careful human coder. Pragmatic remarks (e.g., suggestions that certain global optimizations are possible) as well as explicit consent (e.g., the REORDER attribute of PL/I) are required. Similarly, if programs are to be validated by a program verifier, assistance from the programmer in forming inductive assertions is needed. Communication between the programmer and the optimizer/verifier is by means of ASSUME and ASSERT forms.

An *assumption* is stated by the programmer and is (by and large) believed true by the evaluator. A local assumption

```
ASSUME(X≥0);
```

is taken as true at the point it appears. A global assumption may be extended over some range by means of the infix operator IN, e.g.,

```
ASSUME(X≥0) IN BEGIN...END;
```

where the assumption is to hold over the BEGIN-END block and over all ranges called by that block. The function of an *assumption* is to convey information which the programmer knows is true but which cannot be deduced from the program. Specifications of the well-formedness of input data are assumptions as are statements about the behavior of external procedures analyzed separately.

Assertions, on the other hand, are verifiable. From the program text and the validity of the program's assumptions, it is possible—at least in principle—to validate each assertion. For example,

```
ASSERT(FOR I FROM 1 TO N DO TRUEP(A[I]≥
                              B[I])) IN BEGIN...END
```

* The only significant exception is for arithmetic expressions with no common subexpressions.[10]

should be provably true over the entire BEGIN-END block, given that all program assumptions are correct.

The interpreter, optimizer, and verifier each treat assumptions and assertions in different ways. Since the *interpreter* is used primarily for debugging, it takes the position that the programmer is not to be trusted. Hence, it checks everything, treating assumptions and assertions identically—as extended Boolean expressions to be evaluated and checked for *true* (*false* causing an ERROR and, in general, suspension of the program). Local assertions and assumptions are evaluated in analogy with the conditional expression

NOT ⟨expression⟩⇒ERROR(...)

(This is similar to the use of ASSERT in ALGOL W.[12]) Assumptions and assertions over some range are checked over the entire range. This can be done by checking the validity at the start of the domain and setting up a *condition monitor* (e.g., cf. Reference 13) which will cause a software interrupt if the condition is ever violated during the range.

Hence, in interpreted execution, assumptions and assertions act as comments whose correctness is checked by the evaluator, providing a rather nice debugging tool. Not only are errors explicitly detected by a false assertion, but when errors of other sorts occur (e.g., overflow, data type mismatch, etc.), the programmer scanning through the program is guaranteed that certain assertions were valid for that execution. Since debugging is often a matter of searching the execution path for the least source of an error, certainty that portions of the program are correct is as valuable as knowledge of the contrary.

The *compiler* simply believes assertions and assumptions and uses their validity in code optimization. Consider, for example, the assignment

$$X \leftarrow B[I-J]-60$$

Normally, the code for this would include subscript bounds checking. However, in

$$X \leftarrow (\text{ASSERT}(1 \leq I-J \wedge I-J \leq \text{LENGTH}(B))) \quad \text{IN } B[I-J]-60$$

the assertion guarantees that the subscript is in range and no run-time check is necessary.

While assertions and assumptions are handled by the compiler in rather the same way, there are a few differences. Assumptions are the more powerful in that they can be used to express knowledge of program behavior which could not be deduced by the compiler, either because necessary information is not available (e.g., facts about a procedure which will be input during program execution) or because the effort of deduction is prohibitive (e.g., the use of deep results of number theory in a program acting on integers). Separate compilation makes the statement of such assumptions essential, e.g.,

ASSUME(SAFE(P)) IN BEGIN...END

insures that the procedure P is free of side effects and hence can be subject to common subexpression elimination.

Unlike assumptions, assertions can be generated by the compiler as logical consequences of assumptions, other assertions, and the program text. Consider, for example, the following conditional block (cf. Appendix for syntax), where L is a pointer to a list structure.

BEGIN L=NIL⇒...; ...CDR(L)...END

Normally, the CDR operation would require a check for the empty list as an argument. However, provided that there are no intervening assignments to L, the compiler may rewrite this as

BEGIN L=NIL⇒...; ASSERT(L≠NIL)
IN BEGIN...CDR(L)...END END

in which case no checks are necessary. Assertions added by the compiler and included in an augmented source listing provide a means for the compiler to record its deductions and explicitly transmit these to the programmer.

The program *verifier* treats assumptions and assertions entirely differently. Assumptions are believed.* Assertions are to be proved or disproved[14,15] on the basis of the stated assumptions, the program text, the semantics of the programming language, and specialized knowledge about the subject data types. In the case of integers, there has been demonstrable success—the assertion verifier of King has been applied successfully to some definitely non-trivial algorithms. Specialized theorem provers for other domains may be constructed. Fortunately, the number of domains is small. In ALGOL 60, for example, knowledge of the reals, the integers, and Boolean expressions together with an understanding of arrays and array subscripting will handle most program assertions.

In an extensible language, the situation is more complex, but not drastically so. The base language data types are typically those of ALGOL 60 plus a few others, e.g., characters; the set of formation rules for data aggregates consists of arrays, plus structures and pointers.

* One might, conceivably, check the internal consistency of a set of assumptions, i.e., test for possible contradictions.

Only the treatment of pointers presents any new issues—these because pointers allow data sharing and hence access to a single entity under a multiplicity of names (i.e., access paths). This is analogous to the problem of subscript identification, but is compounded since the access paths may be of arbitrary length. However, recent work[16] shows promise of providing proof techniques for pointers and structures built of linked nodes. Since all extension sets ultimately derive their semantics from the base language, it suffices to give a formal treatment to the primitive modes and the built-in set of formation rules—assertions on all other modes can be mapped into and verified on the base.**

One variation on the program verifier is the *notifier*. Whereas the verifier uses formal proof techniques to certify correctness, the notifier uses relatively unsophisticated means to provide counterexamples. One can safely assume that most programs will not be initially correct; hence, substantial debugging assistance can be provided by simply pointing out errors. This can be done somewhat by trial and error—generating values which satisfy the assumptions and running the program to check the assertions. Since programming errors typically occur at the extremes in the space of data values, a few simple heuristics may serve to produce critical counterexamples. If, as appears likely, the computation time for program verification is considerable, the use of a simple, quicker means to find the majority of bugs will be of assistance on online program production. While the notifier can never validate programs, it may be helpful in creating them.

OPTIMIZATION, EXTENSION SETS, AND TRANSFORMATION SETS

One of the advantages of an extensible language over a special purpose language developed to handle a new application arises from the economics of optimization. In an extensible language system, each extended language L_i is defined by an extension set E_i in terms of the base language. Since there is only a single base, one can afford to spend considerable effort in developing optimization techniques for it. Algorithms for register allocation, common subexpression detection, elimination of variables, removal of computation from loops, loop fusion, and the like need be developed and programmed only once. All extensions will take advantage of these. In contrast, the compiler for each special purpose language must have these optimizations explicitly included. This is already a reasonably large programming project, so large that many special purpose languages go essentially unoptimized. As the set of known optimization techniques grows, the economic advantage of extensible language optimization will increase.

There is one flaw in the above argument, which we now repair. There is the tacit assumption that all optimization properties of an extended language L_i can be obtained from the semantics and pragmatics of the base. While the logical dependency is strictly valid, taking this as a complete technique is rather impractical. While certain optimization properties—those concerned solely with control and data flow—can be well optimized in terms of the base language, other properties depending on long chains of reasoning would tax any optimizer that sought to derive them every time they were required.

The point, and our solution, may best be exhibited with an example. Consider

```
FOO(SUBSTRING(I, J, X CONCAT Y))
```

which calls procedure FOO with the substring consisting of the Ith to (I+J−1)th characters of the string obtained by concatenating the contents of string variable X with string variable Y. In an extensible language, SUBSTRING and CONCAT are defined procedures which operate on STRINGs (defined to be ARRAYs of CHARacters).

```
SUBSTRING←
EXPR(I,J:INT, S:STRING; STRING)
BEGIN
  DECL SS:STRING SIZE J;
  FOR K TO J DO SS[K]←S[I+K−1];
  SS
END

CONCAT←
EXPR(A,B:STRING; STRING)
BEGIN
  DECL R:STRING SIZE LENGTH(A)+
    LENGTH(B);
  FOR M TO LENGTH(A) DO R[M]←A[M];
  FOR M TO LENGTH(B) DO R[M+LENGTH(A)]
    ←B[M];
  R
END
```

One could compile code for the above call on FOO by compiling three successive calls—on CONCAT,

** This gives only a formal technique for verification, i.e., specifies what must be axiomatized and gives a valid reduction technique. It may well turn out that such reduction is not a practical solution if the resulting computation costs are excessive. In such cases, one can use the underlying axiomatization as a basis for deriving rules of inference on an extension set. These may be introduced in a fashion similar to the specialized transformation sets discussed in the next section.

SUBSTRING, and FOO. However, by taking advantage of the properties of CONCAT and SUBSTRING, one can do far better. Substituting the definition of CONCAT in SUBSTRING procedures

```
SUBSTRING(I, J, A CONCAT B)=
BEGIN
  DECL SS:STRING SIZE J;
  DECL S:STRING BYVAL
    BEGIN
      DECL R:STRING SIZE LENGTH(A)
        +LENGTH(B);
      FOR M TO LENGTH(A) DO R[M]←A[M];
      FOR M TO LENGTH(B) DO
        R[M+LENGTH(A)]←B[M];
      R
    END;
  FOR K TO J DO SS[K]←S[I+K−1];
  SS
END
```

The block which computes R may be opened up so that its declarations and computation occur in the surrounding block. Then, since S is identical to R, S may be systematically replaced by R and the declaration for S deleted.

```
BEGIN
  DECL SS:STRING SIZE J;
  DECL R:STRING SIZE LENGTH(A)+
    LENGTH(B);
  FOR M TO LENGTH(A) DO R[M]←A[M];
  FOR M TO LENGTH(B) DO
    R[M+LENGTH(A)]←B[M];
  FOR K TO J DO SS[K]←R[I+K−1];
  SS
END
```

This implies that R[M] is defined by the conditional block

```
BEGIN
  M≤LENGTH(A)⇒A[M];
  B[M−LENGTH(A)]
END
```

Replacing M by I+K−1 and substituting, the assignment loop becomes

```
FOR K TO J DO SS[K]←BEGIN
                      K≤LENGTH(A)−I
                        +1⇒A[I+K−1];
                      B[I+K−LENGTH
                        (A)−1]
                    END
```

Distributing the assignment to inside the block, this has the form

```
FOR x TO $v_0$ DO BEGIN
          x≤$v_1$⇒$f_1$(x);
          $f_2$(x)
        END
```

where v_i are loop-independent values and f_i are functions in x. A basic optimization on the base language transforms this into the equivalent form which avoids the test

```
FOR x TO MIN($v_0$,$v_1$) DO $f_1$(x);
FOR x FROM MIN($v_0$, $v_1$)+1 TO $v_0$ DO $f_2$(x);
```

Hence, SUBSTRING(I, J, A CONCAT B) may be computed by a call on the procedure*

```
EXPR(I,J:INT, A,B:STRING; STRING)
BEGIN
  DECL SS:STRING SIZE J;
  FOR K TO MIN(J, LENGTH(A)−I+1) DO
    SS[K]←A[I+K−1];
  FOR K FROM MIN(J, LENGTH(A)−I+1)+1
    TO J DO SS[K]←B[I+K−LENGTH(A)−1];
  SS
END
```

This could, in principle, be deduced by a compiler from the definitions of SUBSTRING and CONCAT. However, there is no way for the compiler to know *a priori* that the substitution has substantial payoff. If the expression SUBSTRING(I,J,A CONCAT B) were a critical unit operation, the heuristic "try all possible compilation techniques on key expressions" would discover it. However, the compiler cannot afford to try all function pairs appearing in the program in the hope that some will simplify—the computational cost is too great. Instead, the programmer specifies to the compiler the set of transformations (cf. Reference 17 for related techniques) he knows will have payoff.

```
TRANSFORM(I,J:INT, X,Y:STRING;
        SUBSTITUTE)
      SUBSTRING(I, J, X CONCAT Y)
TO
      SUBSTITUTE(Z:X CONCAT Y,
        SUBSTRING(I,J,Z)) (I, J, X, Y)
```

In general, a transformation rule has the format

```
TRANSFORM(⟨pattern variables⟩; ⟨action variables⟩)
      ⟨pattern⟩
TO
      ⟨replacement⟩
```

* Normal common subexpression elimination will recognize that LENGTH(A), I-1, and MIN(J, LENGTH(A)-I+1) need be calculated only once.

All lexemes in the pattern and replacement are taken literally except for the ⟨pattern variables⟩ and ⟨action variables⟩. The former are dummy arguments, statement-matching variables, etc.; the latter denote values used to derive the actual transformation from the input transformation schemata. In the above case, the procedure SUBSTITUTE is called to expand CONCAT within SUBSTRING as the third argument. The simplified result, $\mathcal{P}$, is applied to the dummy arguments. Hence, calls such as SUBSTRING(3,2*N+C, AA CONCAT B7) are transformed into calls on $\mathcal{P}$(3,2*N+C, AA, B7).

When defining an extension set, the programmer defines the unit data types, unit operations, and additionally the significant transformations on the problem domain. These *domain-dependent transformations* are adjoined to the set of *base transformations* to produce the total transformation set. The program, as written, specifies the function to be computed; the transformation set provides an orthogonal statement of how the computation is to be optimized.

For example, in adding a string manipulation extension, one would first define the data type STRING (fixed length array of characters). Next, one defines the unit operations: LENGTH, CONCATenate, SUBSTRING, SEARCH (for a string x as part of a string y starting at position i and return the initial index or zero if not present). Finally, one defines the transformations on program units involving these operations.

```
TRANSFORM(X,Y:STRING) LENGTH(X
  CONCAT Y)
TO LENGTH(X)+LENGTH(Y)

TRANSFORM(A,X,Y,Z:STRING; SUBSTITUTE)
  X CONCAT Y CONCAT Z
TO SUBSTITUTE(A: Y CONCAT Z;
  X CONCAT A) (X,Y,Z)
```

So long as the transformations are entirely local, they act only as macro replacements. The significant transformations in an extension set are those which make global, far reaching changes to program or data. Clearly, these changes will require knowledge, assumed or asserted, about that portion of the program affected by these changes.

Consider, for example, the issue of string variables in the proposed extension set. If a string variable is to have a fixed capacity, the type STRING is satisfactory. If variable capacity is desired but an upper bound can be established for each string variable, the type VARSTRING could be defined like string VARYING in PL/I. If completely variable capacity is required, a string variable would be implemented as a pointer to a simple STRING (i.e., PTR(STRING)) with the understanding that assignment of a string value to such a string variable causes a copy of the string to be made and the pointer set to address the copy.* With these three possible representations available, one would define the data type *string variable* to be

```
ONEOF(STRING, VARSTRING, PTR(STRING))
```

Each string variable is one of these three data types. To provide for the worst case, the programmer could specify each formal parameter string variable to be ONEOF(STRING, VARSTRING, PTR(STRING)) and specify each local string variable to be a PTR-(STRING). A program so written would be correct, but its performance would, in general, suffer from unused generality. Each string variable whose length is fixed can be redeclared

```
TRANSFORM(D1,D2:DECLIST, S:STATLIST,
    F:FORM, X; WHEN)
  WHEN (CONSTANT(LENGTH(X))) IN
    BEGIN D1; DECL X:PTR(STRING)
    BYVAL F; D2; S END
TO
  BEGIN D1; DECL X:STRING BYVAL F;
    D2; S END
```

The predicate WHEN appearing in a pattern is handled in somewhat the same fashion as are ASSERTions during program verification. It is proved as part of the pattern matching; the transformation is applicable only if the predicate is provably TRUE and the literal part of the pattern matches. Here, it must be proved that LENGTH(X) is a constant over the block B and all ranges called by B. If so, the variable can be of type STRING. Similarly, if there is a computable

* This does not exhaust the list of possible representations for strings. To avoid copying in concatenation, insertion, and deletion, one could represent strings by linked lists of characters nodes: each node consisting of a character and a pointer to the next node. A string variable could then be a pointer to such node lists. To minimize storage, one could employ hashing to insure that each distinct sequence of characters is represented by a unique string-table-entry; a string variable could then be a pointer to such string-table-entries. Hashing and implementing strings by linked lists could be combined to yield still another representation of strings. In the interest of brevity, we consider only three rather simple representations; however, the point we make is all the stronger when additional representations are considered.

maximum length less than a reasonable upper limit LIM, then the data type VARSTRING can be used.

```
TRANSFORM(D1,D2:DECLIST, B:BLOCK,
     F:FORM, K:INT, X; WHEN)
   BEGIN D1; DECL X:PTR(STRING)
     BYVAL F; D2; WHEN(LENGTH(X)≤
     K∧K≤LIM) IN B
   END
TO
   BEGIN D1; DECL X:VARSTRING SIZE
     K BYVAL F; D2; B END
```

To prove an assertion for a variable X over some range, it suffices to prove the assertion true of all expressions that are assignable to X in that range. An assertion about LENGTH(X) is reasonable to validate since it entails only theorem proving over the integers[18]—once the string manipulation routines are reinterpreted as operations on string lengths. Fortunately, most of the interesting predicates are of this order of difficulty. Typical WHEN conditions are: (1) a variable (or certain fields of a data structure) is not changed; (2) an object in the heap is referenced only from a given pointer; (3) whenever control reaches a given program point, a variable always has (or never has) a given value (or set of values); (4) certain operations are never performed on certain elements of a data structure. Such conditions are usually easier to check than those concerned with correct program behavior, since only part of the action carried out by the algorithm is relevant.

That is, the technique suggested above for simplifying proofs about string manipulation operators by considering only string lengths generalizes too many related cases. To verify a predicate concerned with certain properties, one takes a valuation of the program on a model chosen to abstract those properties.[19] The program is run by a special interpreter which performs the computation on the simpler data space tailored to the property. To correct for the loss of information (e.g., the values of most program tests are not available), the computation is conservative (e.g., the valuation of a conditional takes the union of the valuations of the possible arms). If the valuation in the model demonstrates the proposition, it is valid for the actual data space. While this is a sufficient condition, not a necessary one, an appropriate model should seldom fail to prove a true proposition.

CONCLUSION

An interpreter, a compiler, a source-level optimizer employing domain-specific transformations, and a program verifier each compute a valuation over some model. Fitting these valuators together so as to exploit the complementarity of their models is a central task in constructing a powerful programming tool.

ACKNOWLEDGMENT

The author would like to thank Glenn Holloway and Richard Stallman for discussions concerning various aspects of this paper.

REFERENCES

1 B WEGBREIT
The ECL programming system
Proc AFIPS 1971 FJCC Vol 39 AFIPS Press Montvale New Jersey pp 253-262

2 A J PERLIS
The synthesis of algorithmic systems
JACM Vol 17 No 1 January 1967 pp 1-9

3 T E CHEATHAM et al
On the basis for ELF—an extensible language facility
Proc AFIPS FJCC 1968 Vol 33 pp 937-948

4 D G BOBROW
Requirements for advanced programming systems for list processing
CACM Vol 15 No 7 July 1972

5 T E CHEATHAM B WEGBREIT
A laboratory for the study of automating programming
Proc AFIPS 1972 SJCC Vol 40

6 W TEITELMAN et al
BBN-LISP
Bolt Beranek and Newman Inc Cambridge Massachusetts July 1971

7 E W DIJKSTRA
Recursive programming
Numerische Mathematik 2 (1960) pp 312-318. Also in Programming Systems and Languages S Rosen (Ed) McGraw-Hill New York 1967

8 J MOSES
The function of FUNCTION in LISP
SIGSAM Bulletin July 1970 pp 13-27

9 IBM SYSTEM/360
PL/I language reference manual
Form C28-8201-2 IBM 1969

10 R SETHI J D ULLMAN
The generation of optimal code for arithmetic expressions
JACM Vol 17 No 4 October 1970 pp 715-728

11 A V AHO J D ULLMAN
Transformations on straight line programs
Conf Rec Second Annual ACM Symposium on Theory of Computing SIGACT May 1970 pp 136-148

12 R L SITES
Algol W reference manual
Technical Report CS-71-230 Computer Science Department Stanford University August 1971

13 D G BOBROW B WEGBREIT
A model and stack implementation of multiple environments
Report No 2334 Bolt Beranek and Newman Cambridge Massachusetts March 1972 submitted for publication

14 R F FLOYD
Assigning meanings to programs
Proc Symp Appl Math Vol 19 1967 pp 19-32

15 R F FLOYD
Toward interactive design of correct programs
Proc IFIP Congress 1971 Ljubljana pp 1-5

16 J POUPON B WEGBREIT
Verification techniques for data structures including pointers
Center for Research in Computing Technology Harvard University in preparation

17 B A GALLER A J PERLIS
A proposal for definitions in Algol
CACM Vol 10 No 4 April 1967 pp 204-219

18 J C KING
A program verifier
PhD Thesis Department of Computer Science
Carnegie-Mellon University September 1969

19 M SINTZOFF
Calculating properties of programs by valuations on specific models
SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 January 1972 pp 203-207

20 B WEGBREIT et al
ECL programmer's manual
Center for Research in Computing Technology Harvard University Cambridge Massachusetts January 1972

APPENDIX: A BRIEF DESCRIPTION OF EL1 SYNTAX

To a first approximation, the syntax of EL1 is like that of ALGOL 60 or PL/I. Variables, subscripted variables, labels, arithmetic and Boolean expressions, assignments, gotos and procedure calls can all be written as in ALGOL 60 or PL/I. Further, EL1 is—like ALGOL 60 or PL/I—a block structured language. Executable statements in EL1 can be grouped together and delimited by BEGIN END brackets to form blocks. New variables can be created within a block by declaration; the scope of such variable names is the block in which they are declared.

The syntax of EL1 differs from that of ALGOL 60 or PL/I most notably in the form of conditionals, declarations, and data type specifiers. For the purposes of this paper, it will suffice to explain only these points of difference. (A more complete description can be found in Reference 20.)

A.1 *Conditionals*

Conditionals in EL1 are a special case of BEGIN END blocks. In general, each EL1 block has a value—the value of the last statement executed. Normally, this is the last statement in the block. Instead, a block can be conditionally exited with some other value $\mathcal{V}$ by a statement of the form

$$\mathcal{B}\Rightarrow\mathcal{V};$$

If $\mathcal{B}$ is TRUE then the block is exited with the value of $\mathcal{V}$; otherwise, the next statement of the block is executed. For example, the ALGOL 60 conditional

if $\mathcal{B}_1$ *then* $\mathcal{V}_1$ *else if* $\mathcal{B}_2$ *then* $\mathcal{V}_2$ *else* $\mathcal{V}_3$

is written in EL1 as

BEGIN $\mathcal{B}_1\Rightarrow\mathcal{V}_1$; $\mathcal{B}_2\Rightarrow\mathcal{V}_2$; $\mathcal{V}_3$ END

(Unconditional statements of an EL1 block are simply executed sequentially—unless a goto transfers control to a different labeled statement.)

A.2 *Declarations*

The initial statements of a block may be declarations having the format

DECL $\mathcal{L}$: $\mathcal{M}$ $\mathcal{S}$;

where $\mathcal{L}$ is a list of identifiers, $\mathcal{M}$ is the data type, and $\mathcal{S}$ specifies the initialization. For example,

DECL X, Y: REAL BYVAL A[I];

This creates two REAL variables named X and Y and initializes them to separate *copies* of the current value of A[I]. The specification $\mathcal{S}$ may assume one of three forms:

(1) empty—in which case a default initialization determined by the data type is used.
(2) BYVAL $\mathcal{V}$—in which case the variables are initialized to *copies* of the value of $\mathcal{V}$.
(3) SHARED $\mathcal{V}$—in which case the variables are declared to be synonymous with the value of $\mathcal{V}$.

A.3 *Data types*

Built-in data types of the language include: BOOL, CHAR, INT, and REAL. These may be used as data type specifiers to create *scalar* variables.

Array variables may be declared by using the built-in procedure ARRAY. For example,

DECL C: ARRAY(CHAR) BYVAL $\mathcal{V}$;

creates a variable named C which is an ARRAY of

CHARacters. The LENGTH (i.e., number of components) and initial value of C is determined by the value of $\mathcal{V}$.

Procedure—valued variables may be defined by the builtin procedure PROC. For example,

```
DECL G:PROC(BOOL,ARRAY(INT); REAL);
```

declares G to be variable which can be assigned only those procedures which take a BOOL argument and an ARRAY(INT) argument and deliver a REAL result.

A.4 *Procedures*

A procedure may be defined by assigning a procedure value to a procedure-valued variable. For example,

```
IPOWER←
EXPR(X:REAL,N:INT; REAL)
BEGIN DECL R:REAL BYVAL 1; FOR I TO N
  DO R←R*X; R END
```

assigns to IPOWER a procedure which takes two arguments, a REAL and an INT (assumed positive), and computes the exponential.

Automated programmering—The programmer's assistant

by WARREN TEITELMAN*

Bolt, Beranek, & Newman
Cambridge, Massachusetts

INTRODUCTION

This paper describes a research effort and programming system designed to facilitate the production of programs. Unlike automated *programming*, which focuses on developing systems that *write* programs, automated *programmering* involves developing systems which automate (or at least greatly facilitate) those tasks that a programmer performs other than writing programs: e.g., repairing syntactical errors to get programs to run in the first place, generating test cases, making tentative changes, retesting, undoing changes, reconfiguring, massive edits, et al., plus repairing and recovering from mistakes made during the above. When the system in which the programmer is operating is cooperative and helpful with respect to these activities, the programmer can devote more time and energy to the task of programming itself, i.e., to conceptualizing, designing and implementing. Consequently, he can be more ambitious, and more productive.

BBN-LISP

The system we will describe here is embedded in BBN-LISP. BBN-LISP, as a programming *language*, is an implementation of LISP, a language designed for list processing and symbolic manipulation.[1] BBN-LISP as a programming *system*, is the product of, and vehicle for, a research effort supported by ARPA for improving the programmer's environment.** The term "environment" is used to suggest such elusive and subjective considerations as ease and level of interaction, forgivingness of errors, human engineering, etc.

Much of BBN-LISP was designed specifically to enable construction of the type of system described in this paper. For example, BBN-LISP includes such features as complete compatibility of compiled and interpreted code, "visible" variable bindings and control information, programmable error recovery procedures, etc. Indeed, at this point the two systems, BBN-LISP and the programmer's assistant, have become so intertwined (and interdependent), that it is difficult, and somewhat artificial, to distinguish between them. We shall not attempt to do so in this paper, preferring instead to present them as one integrated system.

BBN-LISP contains many facilities for assisting the programmer in his non-programming activities. These include a sophisticated structure editor which can either be used interactively or as a subroutine; a debugging package for inserting conditional programmed interrupts around or inside of specified procedures; a "prettyprint" facility for producing structured symbolic output; a program analysis package which produces a tree structured representation of the flow of control between procedures, as well as a concordance listing indicating for each procedure the procedures that call it, the procedures that it calls, and the variables it references, sets, and binds; etc.

Most on-line programming systems contain similar features. However, the essential difference between the BBN-LISP system and other systems is embodied in the philosophy that the user addresses the system through an (active) intermediary agent, whose task it is to collect and save information about what the user and his programs are doing, and to utilize this information to assist the user and his programs. This intermediary is called the programmer's assistant (or p.a.).

THE PROGRAMMER'S ASSISTANT

For most interactions with the BBN LISP system, the programmer's assistant is an invisible interface between the user and LISP: the user types a request, for example, specifying a function to be applied to a set of arguments. The indicated operation is then per-

* The author is currently at Xerox Palo Alto Research Center 3180 Porter Drive, Palo Alto, California 94304.

** Earlier work in this area is reported in Reference 2.

formed, and a resulting value is printed. The system is then ready for the next request. However, in addition, in BBN-LISP, each input typed by the user, and the value of the corresponding operation, are automatically stored by the p.a. on a global data structure called the *history list.*

The history list contains information associated with each of the individual "events" that have occurred in the system, where an event corresponds to an individual type-in operation. Associated with each event is the input that initiated it, the value it yielded, plus other information such as side effects, messages printed by the system or by user programs, information about any errors that may have occurred during the execution of the event, etc. As each new event occurs, the existing events on the history list are aged, with the oldest event "forgotten".*

The user can reference an event on the history list by a pattern which is used for searching the history list, e.g., FLAG:←$ refers to the last event in which the variable FLAG was changed by the user; by its relative event number, e.g. -1 refers to the most recent event, -2 the event before that, etc., or by an absolute event number. For example, the user can retrieve an event in order to REDO a test case after making some program changes. Or, having typed a request that contains a slight error, the user may elect to FIX it, rather than retyping the request in its entirety. The USE command provides a convenient way of specifying simultaneous substitutions for lexical units and/or character strings, e.g., USE X FOR Y AND + FOR *. This permits after-the-fact parameterization of previous events.

The p.a. recognizes such requests as REDO, FIX, and USE as being directed to *it,* not the LISP interpreter, and executes them directly. For example, when given a REDO command, the p.a. retrieves the indicated event, obtains the input from that event, and treats it exactly as though the user had typed it in directly. Similarly, the USE command directs the p.a. to perform the indicated substitutions and process the result exactly as though it had been typed in.

The p.a. currently recognizes about 15 different commands (and includes a facility enabling the user to define additional ones). The p.a. also enables the user to treat several events as a single unit, (e.g. REDO 47 THRU 51), and to name an event or group of events, e.g., NAME TEST -1 AND -2. All of these capabilities allow, and in fact encourage, the user to construct complex *console* operations out of simpler ones in much the same fashion as programs are constructed, i.e., simpler operations are checked out first, and then combined and rearranged into large ones. The important point to note is that the user does *not* have to prepare in advance for possible future (re-) usage of an event. He can operate straightforwardly as in other systems, yet the information saved by the p.a. enables him to implement his "after-thoughts."

UNDOING

Perhaps the most important after-thought operation made possible by the p.a. is that of *undoing* the side-effects of a particular event or events. In most systems, if the user suspects that a disaster might result from a particular operation, e.g., an untested program running wild and chewing up a complex data structure, he would prepare for this contingency by saving the state part of or all of his environment before attempting the operation. If anything went wrong, he would then back up and start over. However, saving/dumping operations are usually expensive and time-consuming, especially compared to a short computation, and are therefore not performed that frequently. In addition, there is always the case where disaster strikes as a result of a supposedly debugged or innocuous operation. For example, suppose the user types

```
FOR X IN ELTS REMOVE PROPERTY
         'MORPH FROM X
```

which removes the property MORPH from every member of the list ELTS, and then realizes that he meant to remove this property from the members of the list ELEMENTS instead, and has thus destroyed some valuable information.

Such "accidents" happen all too often in typical console sessions, and result in the user's either having to spend a great deal of effort in reconstructing the inadvertently destroyed information, or alternatively in returning to the point of his last back-up, and then repeating all useful work performed in the interim. (Instead, using the p.a., the user can recover by simply typing UNDO, and then perform the correct operation by typing USE ELEMENTS FOR ELTS.)

The existence of UNDO frees the user from worrying about such oversights. He can be relaxed and confident in his console operations, yet still work rapidly. He can even experiment with various program and data configurations, without necessarily thinking through all the implications *in advance.* One might argue that this would promote sloppy working habits. However, the same argument can be, and has been, leveled against interactive systems in general. In fact, freeing the user from such details as having to anticipate all of the consequences of an (experimental) change usually re-

* The storage used in its representation is then reusable.

sults in his being able to pay more attention to the conceptual difficulties of the problem he is trying to solve.

Another advantage of undoing as it is implemented in the programmer's assistant is that it enables events to be undone *selectively*. Thus, in the above example, if the user had performed a number of useful modifications to his programs and data structures before noticing his mistake, he would not have to return to the environment extant when he originally typed FOR X IN ELTS REMOVE PROPERTY 'MORPH FROM X, in order to UNDO that event, i.e., he could UNDO this event without UNDOing the intervening events.* This means that even if we eliminated efficiency considerations and assumed the existence of a system where saving the entire state of the user's environment required insignificant resources and was automatically performed before every event, there would still be an advantage to having an undo capability such as the one described here.

Finally, since the operation of undoing an event itself produces side effects, it too is undoable. The user can often take advantage of this fact, and employ strategies that use UNDO for desired operation reversals, not simply as a means of recovery in case of trouble. For example, suppose the user wishes to interrogate a complex data structure in each of two states while successively modifying his programs. He can interrogate the data structure, change it, interrogate it again, then undo the changes, modify his programs, and then repeat the process using successive UNDOs to flip back and forth between the two states of the data structure.

IMPLEMENTATION OF UNDO**

The UNDO capability of the programmer's assistant is implemented by making each function that is to be undoable save on the history list enough information to enable reversal of its side effects. For example, when a list node is about to be changed, it and its original contents are saved; when a variable is reset, its binding (i.e., position on the stack) and its current value are saved. For each primitive operation that involves side effects, there are two separate functions, one which always saves this information, i.e., is always undoable, and one which does not.

Although the overhead for saving undo information is small, the user may elect to make a particular operation *not* be undoable if the cumulative effect of saving the undo information seriously degrades the overall performance of a program because the operation in question is repeated so often. The user, by his choice of function, specifies which operations are undoable. In some sense, the user's choice of function acts as a declaration about frequency of use versus need for undoing. For those cases where the user does not want certain functions undoable once his program becomes operational, but does wish to be able to undo while debugging, the p.a. provides a facility called TESTMODE. When in TESTMODE, the undoable version of each function is executed, regardless of whether the user's program specifically called that version or not.

Finally, all operations involving side effects that are *typed-in* by the user are automatically made undoable by the p.a. by substituting the corresponding undoable function name(s) in the expression before execution. This procedure is feasible because operations that are typed-in rarely involve iterations or lengthy computations *directly*, nor is efficiency usually important. However, as a precaution, if an event occurs during which more than a user-specified number of pieces of undo information are saved, the p.a. interrupts the operation to ask the user if he wants to continue having undo information saved.

AUTOMATIC ERROR CORRECTION—THE DWIM FACILITY

The previous discussion has described ways in which the programmer's assistant is *explicitly* invoked by the user. The programmer's assistant is also automatically invoked by the system when certain error conditions are encountered. A surprisingly large percentage of these errors, especially those occurring in type-in, are of the type that can be corrected without any knowledge about the purpose of the program or operation in question, e.g., misspellings, certain kinds of syntax errors, etc. The p.a. attempts to correct these errors, using as a guide both the context at the time of the error, and information gathered from monitoring the user's requests. This form of *implicit* assistance provided by the programmer's assistant is called the DWIM (*D*o-*W*hat-*I*-*M*ean) capability.

For example, suppose the user defines a function for computing N factoral by typing

```
DEFIN[((FACT (N) IF N=0 THEN 1 ELSE
          NN*(FACT N-1)*].
```

When this input is executed, an error occurs because DEFIN is not the name of a function. However, DWIM

* Of course, he could UNDO all of the intervening events as well, e.g., by typing UNDO THRU ELTS.

** See Reference 1, pp. 22.39–43, for a more complete description of undoing.

* In BBN-LISP] automatically supplies enough right parentheses to match back to the last [.

notes that DEFIN is very close to DEFINE, which is a likely candidate in this context. Since the error occurred in type-in, DWIM proceeds on this assumption, types =DEFINE to inform the user of its action, makes the correction and carries out the request. Similarly if the user then types FATC (3) to test out his function, DWIM would correct FATC to FACT.

When the function FACT is called, the evaluation of NN in NN*(FACT N−1) causes an error. Here, DWIM is able to guess that NN probably means N by using the contextual information that N is the name of the argument to the function FACT in which the error occurred. Since this correction involves a user *program*, DWIM proceeds more cautiously than for corrections to user type-in: it informs the user of the correction it is about to make by typing NN(IN FACT)→N ? and then waits for approval. If the user types Y (for YES), or simply does not respond within a (user) specified time interval (for example, if the user has started the computation and left the room), DWIM makes the correction and continues the computation, exactly as though the function had originally been correct, i.e., no information is lost as a result of the error.

If the user types N (for NO), the situation is the same as when DWIM is not able to make a correction (that it is reasonably confident of). In this case, an error occurs, following which the system goes into a suspended state called a "break" from which the user can repair the problem himself and continue the computation. Note that in neither case is any information or partial results lost.

DWIM also fixes other mistakes besides misspellings, e.g., typing eight for "(" or nine for ")" (because of failure to hit the shift key). For example, if the user had defined FACT as

(IF N=0 THEN 1 ELSE NN*8FACT N−1),

DWIM would have been able to infer the correct definition.

DWIM is also used to correct other types of conditions not considered errors, but nevertheless obviously not what the user meant. For example, if the user calls the editor on a function that is not defined, rather than generating an error, the editor invokes the spelling corrector to try to find what function the user meant, giving DWIM as possible candidates a list of user defined functions. Similarly, the spelling corrector is called to correct misspelled edit commands, p.a. commands, names of files, etc. The spelling corrector can also be called by user programs.

As mentioned above, DWIM also uses information gathered by monitoring user requests. This is accomplished by having the p.a., for each user request, "notice" the functions and variables being used, and add them to appropriate spelling lists, which are then used for comparison with (potentially) misspelled units. This is how DWIM "knew" that FACT was the name of a function, and was therefore able to correct FATC to FACT.

TABLE I—Statistics on Usage

Sessions	exec inputs	edit com-mands	undo saves	p.a. com-mands	spelling correc-tions
1.	1422	1089	3418	87	17
2.	454	791	782	44	28
3.	360	650	680	33	28
4.	1233	3149	2430	184	64
5.	302	24	558	8	0
6.	109	55	677	6	1
7.	1371	2178	2138	95	32
8.	400	311	1441	19	57
9.	294	604	653	7	30
10.	102	44	1044	1	4
11.	378	52	1818	2	2

As a result of knowing the names of user functions and variables (as well as the names of the most frequently used system functions and variables), DWIM seldom fails to correct a spelling error the user feels it should have. And, since DWIM knows about common typing errors, e.g., transpositions, doubled characters, shift mistakes, etc.,* DWIM almost never mistakenly corrects an error. However, if DWIM *did* make a mistake, the user could simply interrupt or abort the computation, UNDO the correction (all DWIM corrections are undoable), and repair the problem himself. Since an error had occurred, the user would have had to intervene anyway, so that DWIM's unsuccessful attempt at correction did not result in extra work for him.

STATISTICS OF USE

While monitoring user requests, the programmer's assistant keeps statistics about utilization of its various capabilities. Table I contains 5 statistics from 11 different sessions, where each corresponds to several

* The spelling corrector also can be instructed as to specific user misspelling habits. For example, a fast typist is more apt to make transposition errors than a hunt-and-peck typist, so that DWIM is more conservative about transposition errors with the latter. See Reference 1, pp. 17.20–22 for complete description of spelling corrections.

TABLE II—Further Statistics

exec inputs	3445
undo saves	10394
changes undone	468
calls to editor	387
edit commands	3027
edit undo saves	1669
edit changes undone	178
p.a. commands	360
spelling corrections	74
calls to spelling corrector	1108*
# of words compared	5636**
time in spelling corrector (in seconds)	80.2
CPU time (hr:min:sec)	1:49:59
console time	21:36:48
time in editor	5:23:53

* An "error" may result in several calls to the spelling corrector, e.g., the word might be a misspelling of a break command, of a p.a. command, or of a function name, each of which entails a separate call.

** This number is the actual number of words considered as possible respellings. Note that for each call to the spelling corrector, on the average only five words were considered, although the spelling lists are typically 20 to 50 words long. This number is so low because frequently misspelled words are moved to the front of the spelling list, and because words are not considered that are "obviously" too long or too short, e.g., neither AND nor PRETTYPRINT would be considered as possible respellings of DEFIN.

individual sessions at the console, following each of which the user saved the state of his environment, and then resumed at the next console session. These sessions are from eight different users at several ARPA sites. It is important to note that with one exception (the author) the users did not know that statistics on their session would be seen by anyone, or, in most cases, that the p.a. gathered such statistics at all.

The five statistics reported here are the number of:

1. requests to executive, i.e., in LISP terms, inputs to evalquote or to a break;
2. requests to editor, i.e., number of editing commands typed in by user;
3. units of undo information saved by the p.a., e.g., changing a list node (in LISP terms, a single *rplaca* or *rplacd*) corresponds to one unit of undo information;
4. p.a. commands, e.g., REDO, USE, UNDO, etc.;
5. spelling corrections.

After these statistics were gathered, more extensive measurements were added to the p.a. These are shown for an extended session with one user (the author) in Table II below.

CONCLUSION

We see the current form of the programmer's assistant as a first step in a sequence of progressively more intelligent, and therefore more helpful, intermediary agents. By attacking the problem of representing the intent behind a user request, and incorporating such information in the p.a., we hope to enable the user to be less specific, and the p.a. to draw inferences and take more initiative.

However, even in its present relatively simplistic form, in addition to making life a lot more pleasant for users, the p.a. has had a suprising synergistic effect on user productivity that seems to be related to the *overhead that is involved when people have to switch tasks or levels.* For example, when a user types a request which contains a misspelling, having to retype it is a minor annoyance (depending, of course, on the amount of typing required and the user's typing skill). However, if the user has mentally *already performed that task,* and is thinking ahead several steps to what he wants to do next, then having to go back and retype the operation represents a disruption of his thought processes, in addition to being a clerical annoyance. The disruption is even more severe when the user must also repair the damage caused by a faulty operation (instead of being able to simply UNDO it).

The p.a. acts to minimize these distractions and diversions, and thereby, as Bobrow puts it, "... greatly facilitates construction of complex programs because it allows the user to remain thinking about his program operation at a relatively high level without having to descend into manipulation of details."[3] We feel that similar capabilities should be built into low level debugging packages such as DDT, the executive language of time sharing systems, etc., as well as other "high-level" programming languages, for they provide the user with a significant *mental mechanical advantage* in attacking problems.

REFERENCES

1 W TEITELMAN D G BOBROW A K HARTLEY D L MURPHY
BBN-LISP TENEX reference manual
BBN Report July 1971

2 W TEITELMAN
Toward a programming laboratory
Proceedings of First International Joint Conference on Artificial Intelligence
Washington May 1969

3 D G BOBROW
Requirements for advanced programming systems for list processing (to be published July 1972 CACM)

A programming language for real-time systems

by A. KOSSIAKOFF and T. P. SLEIGHT

The Johns Hopkins University
Silver Spring, Maryland

SUMMARY

This paper describes a different approach to facilitating the design of efficient and reliable large scale computer programs. The direction taken is toward less rather than more abstraction, and toward using the computer most efficiently as a data processing machine. This is done by expressing the program in the form of a two-dimensional network with maximum visibility to the designer, and then converting the network automatically into efficient code. The interactive graphics terminal is a most powerful aid in accomplishing this process. The principal objectives are as follows:

1. Provide a computer-independent representation of a process to be accomplished by a specified (target) computer, and automatically transforming this representation into a complete program, in the assembly language of the specified computer.
2. Design the representation so as to make highly visible the processing and flow of individual data, as well as that of control logic, in the form of a two-dimensional network, and make it understandable to engineers, scientists and computer programmers.
3. Design the representation so that it can be configured readily on an interactive computer-driven graphics terminal.
4. Design a simple but powerful set of computer-independent building blocks, called Data Circuit Elements, for representing the process to be accomplished by a computer using distinct forms to represent each class of function.
5. Enable the user to simulate the execution of the Data Flow Circuits by inputting realistic data and observing the resultant logic and data flow.
6. Facilitate the design of an efficient complex data processing system by making visible the core usage and running time of each section of the process, thus avoiding the construction of a program which exceeds the capacity of the target computer, or which uses undue core capacity and time for low-priority operations.
7. Provide a representation of a computer program which is self-documenting, in a manner clearly understandable by either an engineer or programmer, making clearly visible the interfaces among subunits, the branch points and the successive steps of handling each information input.

INTRODUCTION

The development, "debugging," and maintenance of computer programs for complex data-processing systems is a difficult and increasingly expensive part of modern systems design, especially for those systems which involve high speed real-time processing. The problem is aggravated by the absence of a lucid representation of the operations performed by the program or of its internal and external interfaces. Thus, the successful use of modern digital computers in automating such systems has been severely impeded by the large expenditure of time and money in the design of complex computer programs. The development of software is increasingly regarded as the limiting factor in system development.

The individual operations of the central processing unit of a general purpose digital computer are very elementary, with the result that a relatively long sequence of instructions is required to accomplish most data-processing tasks. For this reason, programming languages have been developed which enable the programmer to write concise higher level instructions. A compiler then translates these high-level instructions into the machine code for a given computer. The programmer's task is greatly facilitated, since much of the detailed housekeeping is done by the compiler.

High level languages are very helpful in designing programs for mathematical analysis and business applications. In contrast, they do not lend themselves to the design of real-time programs for complex automated systems. The high-level languages obscure the relation between instructions and the time required for their execution, and thus can produce a program which later proves to require unacceptably long processing times. Further, automated systems must often accommodate large variations in data volume and "noise" content. The use of existing high-level programming languages inherently obscures the core requirements for storing the code and data. This results in inefficient use of memory and time, by a factor as high as three, and is therefore a limiting factor in data handling capacity. In such systems assembly language is often used to insure that the program meets all system requirements, despite the increased labor involved in the detailed coding. For these reasons the design of computer programs for real-time systems is much more difficult than the preparation of programs for batch-type computational tasks.

An even more basic difficulty is a serious communication gap between the engineers and the programmers. Engineers prepare the design specifications for the program to fit the characteristics of the data inputs and the rate and accuracy requirements of the processed outputs. In so doing they cannot estimate reliably the complexity of the program that will result. The programmers have little discretion in altering the specifications to accommodate the limitations on computer capacity and processing times. Consequently, the development of a computer for an automated system consequently often results in an oversized and unbalanced product after an inordinate expenditure of effort and time.

PRINCIPAL FEATURES

The principal features of the technique developed to solve these problems and the objectives listed in the Summary, are as follows:

*Data flow circuit language**

The basis of the technique is the representation of a computer program in a "language" resembling circuit networks, referred to as Data Flow Circuits. These represent the processing to be done in a form directly analogous to diagrams used by engineers to lay out electronic circuits. Data Flow circuits correspond to a "universal language" having a form familiar to engineers and at the same time translatable directly into computer code. This representation focuses attention on the *flow* of identifiable *data* inputs through alternative paths or "branches" making up a data processing network. The switching of data flow at the branch points of the network is done by signals generated in accordance with required logic. These control signals usually generate "jump" instructions in the computer program.

Data Flow circuits are constructed of building blocks, which will be called Data Circuit Elements, each of which represents an operation equivalent to the execution of a set of instructions in a general-purpose computer. These Data Circuit elements are configured by the designer into a two-dimensional network, or Data Flow circuit, which represents the desired data processing, as if he were laying out an electronic circuit using equivalent hardware functional elements. Special circuit elements can also be assembled and defined by the designer for his own use.

The direct correspondence between individual Data Circuit elements and actual computer instructions makes it possible to assess the approximate time for executing each circuit path and the required core. This permits the designer to balance during the initial design of the circuit, the requirements for accuracy and capacity against the program "costs" in terms of core and running time. This capability can be of utmost importance in high-data-rate real-time systems, using limited memory.

The Data Flow circuit representation also serves as a particularly lucid form of documenting the final derived computer program. It can be configured into a form especially suited for showing the order in which the program executes each function.

Application of computer graphics

The form of the Data Flow circuits and circuit elements is designed to be conveniently represented in a computer-driven graphics terminal, so as to take advantage of its powerful interactive design capability. In this instance, the Data Flow Circuit is designed on the display by selecting, arranging and connecting elements using a light pen, joystick, keyboard or other graphic aid, in a manner similar to that used in computer design of electronic circuits.

As the circuit is being designed, the computer display program stores the circuit description in an "element interconnection matrix" and a data "dictionary".

* The term "Data Flow" has been employed earlier but with quite different objectives than those described in this work. (W. O. Sutherland, "On-Line Graphical Specification of Computer Procedures," PhD thesis, Massachusetts Institute of Technology, January 10, 1966).

This description is checked by the program and any inconsistencies in structure are immediately drawn to the designer's attention.

Transformation into logical form

After the elements and interconnections have been entered into the interactive computer by means of either a graphic or alphanumeric terminal, the computer converts the Data Flow circuit automatically into an Operational Sequence by means of a Transformation program. This orders the operations performed by the circuit elements in the same sequence as they would be serially processed by the computer.

Code generation and simulation

In this step the computer converts the operational sequence into instructions for the interactive computer. The program logic is then checked out by using sample inputs and examining the outputs. Errors or omissions are immediately called to the attention of the designer so that he can modify the faulty connections or input conditions in the circuit on-line. The assembly language instructions for the target computer are then generated.

Integration and testing

The derived program is assembled by the interactive computer with other blocks of the total program and the result is again checked for proper operation. Subsequent modifications to the program are made by calling up the circuit to be altered, and making the changes at the display terminal.

The above steps provide the Graphical Automatic Programming method for designing, documenting and managing an entire complex computer program through the use of Data Circuit language. The result is highly efficient system software which is expected to be produced at a fraction of the time and cost achievable by present methods.

Data circuit elements

In selecting the "building blocks" to be used as the functional elements of Data Flow circuits, each Data Circuit Element was designed to meet the following criteria:

1. It must be sufficiently basic to have wide application in data processing systems.
2. It must be sufficiently powerful to save the designer from excessive detailing of secondary processes.
3. It must have a symbolic form which is simple to represent and meaningful in terms of its characteristic function, but which will not be confused with existing component notation.

The choice and definition of the basic GAP (Graphical Automatic Programming) Data Circuit Elements has evolved as a result of applications to practical problems. Seven classes of circuit elements have been defined, as follows:

SENSE elements test a particular characteristic of a data input and produce one of two outputs according to whether the result of the test was true or false.

OPERATOR elements perform arithmetic or logical operations on a pair of data inputs and produce a data output.

COMPARISON elements test the relative magnitude of two or three data inputs and produce two or three outputs according to the result of the test.

TRANSFER elements bring data in and out of the circuit from files in memory and from external devices.

INTEGRATING elements, which are in effect complex operator elements, collect the sum or product of repeated operations on two variables.

SWITCHING elements set and read flags, index a series of data words, branch a succession of data signals to a series of alternate branches, and perform other branching functions.

ROUTING elements combine, split, and gate the flow of data and control signals, and provide the linkage between the program block represented by a given Data Flow Circuit and other program blocks (circuits) constituting the overall program. Some routing elements do not themselves produce program instructions, but rather modify those produced by the functional elements to which they are connected.

Table I lists the elements presently defined for initial use in the **G**raphical **A**utomatic **P**rogramming language (GAP). These include four SENSE elements, eleven OPERATOR elements, six COMPARISON elements, six TRANSFER elements, fourteen ROUTING elements, three SWITCHING elements, and six INTEGRATING elements. Others found to meet the basic criteria and be widely applicable will be added to the basic vocabulary. Each designer also may define for his own use special-purpose functions as auxiliary elements, so long as they maintain the basic characteristics, i.e., they accurately show data flow and are directly convertible to machine instructions to permit precise time and core equivalency. Most of these can be built up from combinations of the basic elements.

Figure 1 illustrates the symbolic representation of a typical circuit element of each of the seven classes. Solid lines are used for data signals and dashed lines for control signals. Data inputs are denoted by an X, data outputs by a Y, control inputs by a C and control outputs by a J. When the input or output may be either control or data the letters I or O are used. A U simply means unconnected.

In Figure 1 the sample elements are seen to have the following types and numbers of connections:

Element Type	Name	Data Inputs	Control Inputs	Data Outputs	Control Outputs
ROUTING	DATA SPLIT	1	0	2	0
SENSE	BRANCH ON ZERO	1–2	1–0	0–2	2–0
OPERATOR	ADD	2	1–0	1	0–1
COMPARISON	BRANCH ON COMPARE	2–3	1–0	0–3	3–0
TRANSFER	READ FILE	2	2	1	1
SWITCHING	SET BRANCH	0	3	1	0
INTEGRATING	SUM MULTIPLY	2	0	1	0–1

Figure 1—Data flow circuit elements graphical representation

OPERATOR and COMPARISON elements are provided with an optional control input which serves to delay the functioning of the element until the receipt of the control signal from elsewhere in the circuit. The READ FILE and other loop elements have a control input which serves a different purpose, namely to initiate the next cycle of the loop.

At present, the maximum number of connections for any element is eight and for SENSE and OPERATOR elements it is four. Connections, or terminals, are numbered clockwise with 1 at 12 o'clock.

Data preparation

All of the elements described above have either more than one input or more than one output. There are a number of elementary operations which simply alter a data word, thus having a single input and a single output. These operations include masking, shifting, complementing, incrementing and other simple unit processes ordinarily involved in housekeeping manipulations, as for example packing several variables into a single data word or the reverse.

TABLE I—Data Flow Circuit Elements

SENSE

- Branch on Zero
- Branch on Plus
- Branch on Minus
- Branch on Constant

OPERATOR

- Add
- Average
- Multiply
- Subtract
- Divide
- Exponentiate
- And
- Inclusive or
- Exclusive or
- Minimum
- Maximum

COMPARISON

- Branch on Compare
- Branch on Greater
- Branch on Unequal
- Correlate
- Threshold
- Range Gate

TRANSFER

- Read Word
- Write Word
- Read File
- Write File
- Function Table
- Input Data
- Output Data

INTEGRATING

- Sum Add
- Sum Multiply
- Sum Divide
- Sum Exponentiate
- Product Add
- Product Exponentiate

SWITCHING

- Set Branch
- Read Branch
- Index Data

ROUTING

- Linkage Data
- Passive Split
- Data Split
- Control Split
- Linkage Exit
- Passive Junction
- Data Junction
- Control Junction
- Linkage Store
- Data Gate
- Data Pack
- Linkage Entry
- Data Loop
- Control Loop

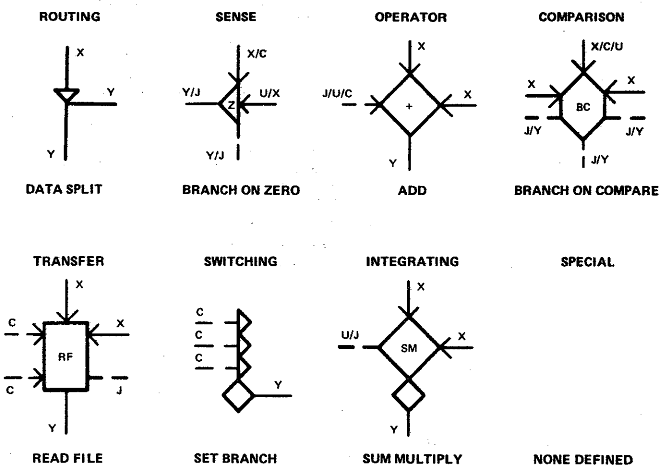


Figure 1—Data flow circuit elements graphical representation

In the Data Flow Circuit notation, such manipulation is specified by a "prepare" operation preliminary to the operation performed by each element. The manipulations involved in data preparation, which represents a major portion of the "housekeeping" labor in programming, are thereafter accomplished automatically along with the translation of the functional operations of the elements in the Data Circuit. This type of operation is designed graphically by closed arrowheads at input terminals.

SAMPLE DATA FLOW CIRCUIT

Description of a sample flow data circuit

The particular system from which the following example has been drawn concerns real-time processing of radar signals or "hits." This function is normally associated with track-while-scan radar systems.

The logic of the example "Hit Sorting Program" illustrated in Figure 2 operates by indexing through a number of hits in the HIT file. Each hit whose amplitude is greater than or equal to a specific threshold (T) is placed in the track (TRK) file. When the HIT file is empty or the TRK file is full the program is exited.

Three functional and seven nonfunctional elements accomplish this task:

1. **R**ead **F**ile (RF), to extract each hit from the HIT file.
2. **B**ranch on **C**ompare (BC), to select hits whose amplitude equals or exceeds the threshold.
3. **W**rite **F**ile (WF), to enter the selected hits into the TRK file for retention.
4. A **D**ata **S**plit (DS), a **D**ata **G**ate (DG), and two **C**ontrol **J**unctions (CJ), distribute the data and control to the correct element terminals. Data inputs to the circuit are provided by a **L**inkage **D**ata element (LD), the control input by a **L**inkage **E**ntry element (LE), and two control exits by a **L**inkage **E**xit element (LX).

In the Data Flow circuit in Figure 2, the numbers in parentheses are unique reference numbers for each element and are prefixed with an "R" in the following text. The reference numbers, R, and element labels in parentheses do not actually appear at the graphic terminals but are used in the explanations of the circuit that follows:

The circuit is activated by a control signal at Read File, element R3.

This element reads out a hit word containing range and amplitude (A). The input at terminal 1 is the base address of the file (HIT) and at terminal 2 is the index (N) for the negative number of hits.

The Data Split (R6) distributes the hit word to the Branch on Compare (R4) and to the Data Gate (R7).

At the Branch on Compare element the amplitude (A) is extracted and used to compare with a threshold (T).

If the amplitude is greater than or equal to

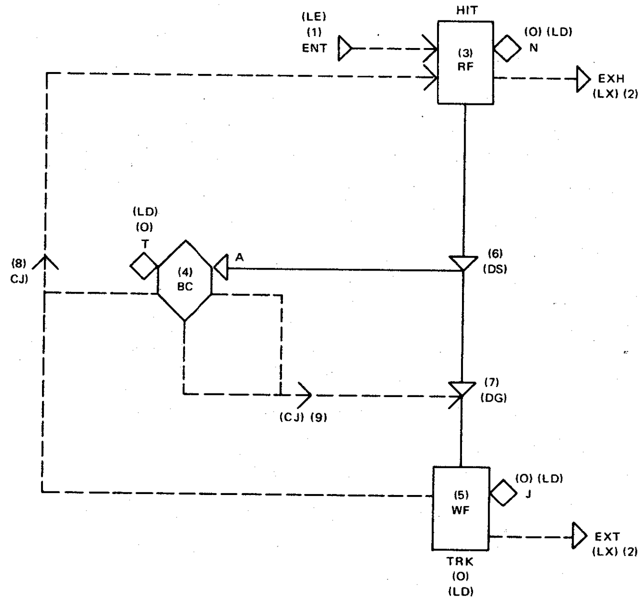


Figure 2—Hit sorting program

the threshold, control is passed to the Data Gate (R7). The original hit word from the Read File enters the Write File element (R5). The index at terminal 2 (J) is incremented. If the index indicates that the file is full the output to the circuit exit is selected. But normally the hit is placed in the file by using the base at terminal 4 (TRK) and the index (J). Control is then passed through the Control Junction (R8) to the looping input (terminal 5) of the Read File.

If the amplitude is less than the threshold, control is immediately passed to the looping input to the Read File.

The looping input to Read File (R3) causes the index (N) to be incremented. If the index indicates that no more entries or hits are present the output to the circuit exit (Linkage Exit) is selected. Otherwise, the next hit is read out and processed through another cycle of the loop.

The quantities HIT, N, T, TRK, and J are outputs from the Linkage Data element (RO). This element does not appear explicitly in the graphical representation, as in the case of the other linkage elements.

The Hit Sorting Program is a simple example for the purpose of explaining the techniques employed in GAP. A circuit more representative in size is the Target Coordinate Computation Circuit, shown in Figure A1 and is developed in an analogous way in the Appendix.

Sample processing of a data flow circuit

Once the particular function has been defined in the form of a GAP circuit on a scratch pad several steps are taken to generate code for the target computer (Figure 3). The circuit is input and checked interactively through a graphics or alphanumeric terminal. The transformation process then converts the two-dimensional circuit representation into a sequential representation of the order in which code for the elements is to be written.

The first step in the transformation is a detailed trace through the circuit. This trace produces a tabulation, called the Operational Sequence. By removing all non-functional steps and other information not necessary for final coding, this is reduced to an ordered list of elements and connections called the Execution Sequence. Each entry in the Execution Sequence corresponds to a dynamic macro statement.

The writing of instructions, Translation, now takes place. The user can select actual target computer code or a simulation of the target computer code. Normally the simulation step is first selected, and later when the circuit has been found to function properly assembly code of the target computer is generated.

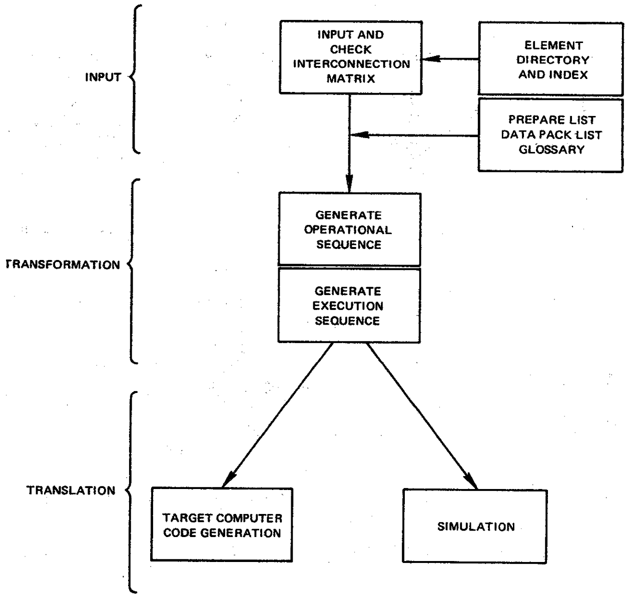

INPUT
TRANSFORMATION
TRANSLATION
INPUT AND CHECK INTERCONNECTION MATRIX
ELEMENT DIRECTORY AND INDEX
PREPARE LIST DATA PACK LIST GLOSSARY
GENERATE OPERATIONAL SEQUENCE
GENERATE EXECUTION SEQUENCE
TARGET COMPUTER CODE GENERATION
SIMULATION

Figure 3—Processing of a GAP circuit

The computer configuration used in this example is an IBM 360/91 operating under MVT. The software is written in the Conversational Programming System (CPS) and is operational under any terminal in the system. The simulation step generates CPS code and the target computer is a Honeywell DDP-516 whose assembly language is called DAP.

Input

The first step in the process of Graphical Automatic Programming is the input of a Data Flow Circuit into an interactive computer terminal. Unless the circuit is very simple, it is usually first laid out roughly on a scratch pad in order to save terminal time during the initial conceptual stages of circuit design. When an alphanumeric terminal is employed, as in the example descr bed in the succeeding sections, the circuit input consists of entering the element labels (e.g., RF for Read File), number of terminals, and Interconnection Matrix. The latter requires the operator to specify only output connections for each element. An interactive program completes the matrix. The output connections are given in terms of the element reference and terminal numbers and type of connection.

Table II gives the Interconnection Matrix of the Hit Sorting Program illustrated in Figure 2. Each row of the matrix lists connections to each of the terminals of the given element. The order of the rows is in accordance with an arbitrary but unique reference number assigned

TABLE II—Interconnection Matrix

	Terminal Number						
R	1	2	3	4	5	6	7
0	C	3Y1	3Y2	4Y2	5Y2	5Y4	1J1
1	0 C7	306					
2	3 I3	5 I3					
3	0 X2	0 X3	2J1	6Y1	8 C3	1 C2	
4	U U	0 X4	9J1	9J2	8J2	6 X3	
5	7 X3	0 X5	2J2	0 X6	U U	8J1	
6	3 X4	7Y1	4Y6				
7	6 X2	9 C3	5Y1				
8	5 C6	4 C5	3J5				
9	4 C3	4 C4	7J2				
—							

to each element. It should be noted again that the linkage elements do not have a graphical representation, but the element terminals to which they are connected are suitably marked and labeled by diamonds or arrowheads.

Each entry in the Interconnection Matrix (Table II) consists of the reference number and terminal number separated by the type of connection. Thus, the Read File at terminal 4, fourth row, fourth column, is connected to element 6 (Data Split) at terminal 1. The

TABLE III—Interconnection Matrix with Element Labels

R	NAME	1	2	3	4	5	6	7
0	LD	C	RFY1	RFY2	BCY2	WFY2	WFY4	LEJ1
1	LE	LDC7	RF06					
2	LX	RFI3	WFI3					
3	RF	LDX2	LDX3	LXJ1	DSY1	CJC3	LEC2	
4	BC	UU	LDX4	*CJJ1	*CJJ2	CJJ2	DSX3	
5	WF	DGX3	LDX5	LXJ2	LDX6	UU	CJJ1	
6	DS	RFX4	DGY1	BCY6				
7	DG	DSX2	*CJC3	WFY1				
8	CJ	WFC6	BCC5	RFJ5				
9	*CJ	BCC3	BCC4	DGJ2				

*CJ with R9.

entry for Data Split (R6) terminal 1 is seen to point directly back to the Read File. The letters specifying the type of connections are as defined earlier.

For the purpose of illustration, Table III is the same Interconnection Matrix with element labels replacing reference numbers.

The Input program checks the forms of terminal connections for a given element by consulting a table called the Element Directory, a sample of which is shown in Table IV. On the left hand side of this table are given the number of terminals and the configuration of the allowed element forms. For example, the configuration of the terminals of the Linkage Data element taken as of the first row of the Interconnection Matrix, CYYYYYJ, is form number 5 of LD in the Element Directory. Mirror image configurations of all forms of most elements are allowed. The Element Directory is accessed via another table, the Element Index, a sample of which is shown in Table V.

In addition to the above inputs, three other files are constructed during the Input process.

1. When elements requiring new data inputs are entered the program requests entry of variable names and formats, which are listed in a Glossary.
2. When Data Pack elements are entered, packing format data are requested by the program and are assembled in the Data Pack List.
3. When a Prepare operation is indicated on an element input its definition is requested and compiled into a Data Prepare list. Multiple prepares at a given input are allowed.

Transformation

The Data Flow Circuit is a two dimensional representation of the flow of data and control. A general purpose digital computer functions with a one-dimensional sequence of instructions**. Thus, the circuit must be "transformed" into a linear sequence of operations. The first step involves tracing the circuit and determining the proper order of operation of each element. The output of this tracing is the Operational Sequence, which is a detailed tabulation of the trace. As stated previously, in the second step a set of dynamic macro statements, the Execution Sequence, is generated from the Operational Sequence. The Translation described in the next section then converts these macro statements into computer instructions.

The algorithm by which the Transformation program carries out a complete trace of the circuit is derived from certain basic transformation rules. These rules are listed in simplified form in Table VI. They differ according to the four basic classes of elements listed in the first column, and also depend on whether or not the connection being traced to an element with two or more inputs is the final input. An element transmits an output, or "executes", only after the final input arrives. In the case of elements with more than one output, such as branching and splitting elements, all but the "direct" output are put in a deferred file. When a non-final input is made to a joining element, a special flag is generated, and the trace continues by taking the latest deferred branching output. When a non-final input is made to a branching or operating element, this fact is flagged and the trace continues at the latest deferred split. In the case of looping elements a special transformation is used.

Operational sequence—For the purpose of illustration, the most important information of the Operational Sequence has been placed in Table VII. The transformation being performed is identified by a Step Number (S) at the extreme left. The Branch Number (BN) separates the circuit into sections or branches having the same logical content. The Link column contains the element reference (R) and terminal (T) of the connection being transformed. The Reference Number (R) specifies which element is currently being transformed and is seen to be the "To" element reference number of the Link in the previous column. The Name identifies the label of the element being transformed. The Transformation Type identifies the transformation procedure to be followed in transforming the element, as described in Table VI. The Input Index (Q), gives the number of additional inputs needed for the element to function. The Direct Output (OT) is the output terminal number through which transformation will normally proceed when the Input Index is zero and the element functions or "executes" (see Table IV). The Deferred Output terminal numbers (DT) specify which outputs of elements are to be deferred when the element functions. The above characteristics of each element are derived from the Element Directory for the specific form of the element used in the circuit.

The first Deferred Link column gives the types of deferred output with S or B referring to splitting or branching elements respectively, and a sequential number to identify each entry for later removal. The "From" portion of the Deferred Link is constructed from the current Reference Number (R) and the Deferred Output terminal number (DT). The "To" portion is found in the Interconnection Matrix. The number of Remaining Deferred Splits (NS) indicates the number of splits currently in the Deferred Link.

** A parallel processor could use a two dimensional input but cannot at this time be considered of general utility.

TABLE IV—Routing Section of Element Directory

ELEMENT DIRECTORY

ROUTING

NO.	LABEL	FORM	NT		Q1	XX	NO	OT	DT	
0	LD	1	3	CYJ	1	00	2	2	3	1/ 7/71
0	LD	2	4	CYYJ	1	0	3	2	3,4	1/ 7/71
0	LD	3	5	CYYYJ	1	0	4	2	3-5	1/ 7/71
0	LD	4	6	CYYYYJ	1	0	5	2	3-6	1/ 7/71
0	LD	5	7	CYYYYYJ	1	0	6	2	3-7	1/ 7/71
0	LD	6	8	CYYYYYYJ	1	0	7	2	3-8	1/ 7/71
1	PS	1	3	XYY	1	1	2	3	2	1/ 7/71
2	DS	1	3	XYY	1	2	2	3	2	1/ 7/71
3	CS	1	3	XJY	1	3	2	3	2	1/ 7/71
3	CS	2	3	CJJ	1	3	2	3	2	1/ 7/71
4	LX	1	1	I	1	4	0	0	0	1/ 7/71
4	LX	2	2	II	2	4	0	0	0	1/ 7/71
4	LX	3	3	III	3	4	0	0	0	1/ 7/71
4	LX	4	4	IIII	4	4	0	0	0	1/ 7/71
4	LX	5	5	IIIII	5	4	0	0	0	1/ 7/71
4	LX	6	6	IIIIII	6	4	0	0	0	1/ 7/71
4	LX	7	7	IIIIIII	7	4	0	0	0	1/ 7/71
4	LX	8	8	IIIIIIII	8	4	0	0	0	1/ 7/71
5	PJ	1	3	XXY	2	5	1	3	0	1/28/71
5	PJ	2	3	CCJ	2	5	1	3	0	1/ 7/71
6	DJ	1	3	XXY	2	6	1	3	0	1/ 7/71
7	CJ	1	3	CCJ	2	7	1	3	0	1/ 7/71
8	LS	1	1	X	1	8	0	0	0	1/ 7/71
8	LS	2	2	XX	2	8	0	0	0	1/ 7/71
8	LS	3	3	XXX	3	8	0	0	0	1/ 7/71
8	LS	4	4	XXXX	4	8	0	0	0	1/ 7/71
8	LS	5	5	XXXXX	5	8	0	0	0	1/ 7/71
8	LS	6	6	XXXXXX	6	8	0	0	0	1/ 7/71
8	LS	7	7	XXXXXXX	7	8	0	0	0	1/ 7/71
8	LS	8	8	XXXXXXXX	8	8	0	0	0	1/ 7/71
10	DG	1	3	XCY	2	10	1	3	0	1/ 7/71
11	DP	1	3	XXY	2	11	1	3	0	1/ 7/71
11	DP	2	4	XXXY	3	11	1	4	0	1/ 7/71
11	DP	3	5	XXXXY	4	11	1	5	0	1/ 7/71
12	LE	1	2	CO	1	12	1	2	0	1/ 7/71
12	LE	2	3	COO	1	12	2	2	3	1/ 7/71
12	LE	3	4	COOO	1	12	3	2	3-4	1/ 7/71
12	LE	4	5	COOOO	1	12	4	3	3-5	1/ 7/71
12	LE	5	6	COOOOO	1	12	5	4	3-6	1/ 7/71
12	LE	6	7	COOOOOO	1	12	6	5	3-7	1/ 7/71
12	LE	7	8	COOOOOOO	1	12	7	2	3-8	1/ 7/71
14	DL	1	4	XXYU	1	17	1	3	0	1/19/71
14	DL	2	4	XXYC	1	18	1	3	0	1/19/71
15	CL	1	4	CCJU	1	19	1	3	0	1/19/71

The final column in Table VII, Next Link, indicates the source of the link to be transformed next and thus the order in which the links in the circuit are subsequently translated into code. It is derived from transformation rules shown in Table VI. The entries S# and B# specify the Next Link to be obtained as a split or branch from the Deferred Link entries. The entry in the Link column in the next step of the Operational Sequence is seen to come from the indicated entry in the Deferred Link column. The legend OT specifies that the Next Link is to originate at the Direct Output of the element being transformed, as given in the R and OT columns of the table. The "From" portion of the Link in the next step is seen to correspond to the R and OT of the previous step; the "To" portion is derived from the Interconnection Matrix.

The following paragraphs illustrate the construction of the Compressed Operational Sequence of Table VII by describing the initialization procedure followed by examples applying each of the transformation rules summarized in Table VI.

Initialization—The sequence is initialized by transforming the Linkage Data element, LD, which by convention is assigned R=0. LD is a special type of split-

TABLE V—Element Index

ELEMENT TYPE	ELEMENT NUMBER	ELEMENT LABEL	NO. FORMS	ELEMENT NAME
ROUTING	0	LD	6	LINKAGE DATA
	1	PS	1	PASIVE SPLIT
	2	DS	1	DATA SPLIT
	3	CS	2	CONTROL SPLIT
	4	LX	8	LINKAGE EXIT
	5	PJ	2	PASSIVE JUNCTION
	6	DJ	1	DATA JUNCTION
	7	CJ	1	CONTROL JUNCTION
	8	LS	8	LINKAGE STORE
	10	DG	1	DATA GATE
	11	DP	3	DATA PACK
	12	LE	7	LINKAGE ENTRY
	14	DL	2	DATA LOOP
	15	CL	1	CONTROL LOOP

ELEMENT TYPE	ELEMENT NUMBER	ELEMENT LABEL	NO. FORMS	ELEMENT NAME
SENSE	17	ZE	11	BRANCH ZERO
	18	NZ	11	BRANCH NOT ZERO
	19	PL	11	BRANCH PLUS
	20	NP	11	BRANCH NOT PLUS
	21	MI	11	BRANCH MINUS
	22	NM	11	BRANCH NOT MINUS
	23	BK	11	BRANCH CONSTANT
	24	NK	11	BRANCH NOT CONSTANT

ting element, deferring all of its outputs as splits S1 to S6, in the Deferred Link column.

Branching (BR)—The first or non-final input (Input Index≠0) to a branching element is received in steps 2, 3, 4, 5 and 6. In every case the Next Link is taken as the oldest split (S1-S5) of the Deferred Link entries.

The arrival of the final input (Input Index=0) is illustrated in steps 8, 10, 16 and 18. The deferred branch outputs are placed in the Deferred Link columns and the Direct Output is normally taken for the Next Link as in steps 8 and 16. The branching alters the logical content of the sequence and begins a new branch as indicated by the Branch Number. If the end of a branch is encountered before all deferred splits have been processed (i.e., NS≠0), then an exception to the normal transformation exists. In this case, Step 10, the Direct Output is placed in the Deferred Link as a branch, and a deferred split, S7, is taken as the Next Link. The looping input of a branching element is another special case in that the element has already functioned previously; therefore, in step 18 the latest deferred branch, B5, is taken as the Next Link.

TABLE VI—Element Transformation Rules

Transformation	Element Function	Transformation Procedure	
		First Input	Final Input
Branching	Branch signal path according to input condition(s)	Store data input(s) Link next split	Defer jump outputs Link to immediate output
Joining	Join two branches	Jump forward to address of output Link next branch	Label address of output Link to output
Operating	Operate on two or more inputs to form an output	Store data input(s) Link next split	Link to output
Splitting	Distribute signal to two elements	(Only one input)	Defer one output Link to other output

TABLE VII—Compressed Operational Sequence

Step No., S	Branch No., BN	Link From - To R, T	Element						Deferred Link		Remaining Deferred Splits, NS	Next Link
			Ref. No., R	Name	Trans. Type	Input Index, Q	Direct Output, OT	Deferred Output(s), DT	Type and No.	From - To R, T		
1	0	- 0, 1	0	LD	SP	$\underline{0}$	(2)	3-7			0	S1
									S1	0, 2 - 3, 1	1	
									S2	0, 3 - 3, 2	2	
									S3	0, 4 - 4, 2	3	
									S4	0, 5 - 5, 2	4	
									S5	0, 6 - 5, 4	5	
									S6	0, 7 - 1, 1	6	
2	0	0, 2 - 3, 1	0	RF	BR	2					5	S2
3	0	0, 3 - 3, 2	3	RF	BR	1					4	S3
4	0	0, 4 - 4, 2	4	BC	BR	1					3	S4
5	0	0, 5 - 5, 2	5	WF	BR	2					2	S5
6	0	0, 6 - 5, 4	5	WF	BR	1					1	S6
7	0	0, 7 - 1, 1	1	LE	OP	$\underline{0}$	2				0	OT
8	1	1, 2 - 3, 6	3	RF	BR	$\underline{0}$	4	3			0	OT
									B1	3, 3 - 2, 1		
9	2	3, 4 - 6, 1	6	DS	SP	$\underline{0}$	3	2			0	OT
									S7	3, 4 - 7, 1	1	
10	2	3, 4 - 4, 6	4	BC	BR	$\underline{0}$	5	3-4			1	S7
									B2	4, 3 - 9, 1	1	
									B3	4, 4 - 9, 2	1	
									B4	4, 5 - 8, 2	1	
11	2	3, 4 - 7, 1	7	DG	OP	1					0	B4
12	3	4, 5 - 8, 2	8	CJ	JN	1					0	B3
13	4	4, 4 - 9, 2	9	CJ	JN	1					0	B2
14	5	4, 3 - 9, 1	9	CJ	JN	$\underline{0}$	3				0	OT
15	6	9, 3 - 7, 2	7	DG	OP	$\underline{0}$	3				0	OT
16	6	7, 3 - 5, 1	5	WF	BR	$\underline{0}$	6	3			0	OT
									B5	5, 3 - 2, 2	0	
17	7	5, 6 - 8, 1	8	CJ	JN	$\underline{0}$	3				0	OT
18	8	8, 3 - 3, 5	3	RF	BR*	4*	4				0	B5
19	9	5, 3 - 2, 2	2	LX	JN	1					0	B1
20	10	3, 3 - 2, 1	2	LX	JN	0					0	Finis

*Looping input

Joining (JN)—These elements end the current branch regardless of which input arrives, but the Next Link does depend upon the arriving input. In steps 12, 13 and 19 where a non-final input arrives, the latest deferred branch becomes the Next Link.

With the final input arriving, as in steps 14, 17 and 20, the Direct Output becomes the Next Link.

Operating (OP)—Operating elements do not in themselves affect the branch status. The Next Link following the arrival of a non-final input is normally taken as a split from the Deferred Link column, but when there is no deferred split as in step 11, the Next Link is taken as a branch from the Deferred Link column.

The final input simply propagates via the Direct Output as in steps 7 and 15.

Splitting (SP)—These elements place the Deferred Output in the Deferred Link file (as in step 9) and proceed to the Direct Output as the Next Link. It will be noted that the "From" connection in both the Deferred Link (S7) and the Next Link (step 11) is not the output of the Data Split element, 6, 2, but rather the output of the previous functional element, 3, 4. This in effect eliminates reference to the DS element in the Operational Sequence since its only function is to provide multiple output connections to a functional element.

A listing of the computer generated Operational Sequence is given in Table VIII. This listing contains much more detail than is necessary to understand the transformation process. All of the information in the Compressed Operational Sequence, Table VII, is found in Table VIII. The items which have specific columns in each table are shown in parentheses in the compressed table. The "To" portion of the Link of Table VII is labeled LIRT in Table VIII (I for Input). The "From" portion is labeled either LORT (O for Output) or LSRT (S for Store), or both, depending on whether the output has or has not been stored. The Transformation Type is a simplification of the Transformation Number, X, in Table VIII. The Deferred Link is constructed from −IRT, −ORT and −SRT in a manner analogous to the Link. The Type of Deferred Link is a simplification of the F entry, Splits being classed as Control (C) or Data (D) and Branches as normal Branches (B) or Next branches (N) in Table VIII. The Next Link does not appear in the computer generated Operational Sequence but has been added to the compressed form for illustration.

Columns appearing in Table VIII not referred to in Table VII are: XEN- element identification number; NB- number of deferred branches; LN- and LSN-

TABLE VIII—Operational Sequence

S	R	XEN	Q	X	OT	DT	F	-SRT	-ORT	-IRT	LORT	LSRT	LIRT	NS	NB	BN	LN	LSN	LB
1	0	0	0	0	2	3					000	000	001	0	0	0	0	0	0
							D	0 2		31									
							D	0 3		32									
							D	0 4		42									
							D	0 5		52									
							D	0 6		54									
							C		0 7	11									
2	3	67	2	21	4	3					000	0 2	31	5	0	0	0	0	4
3	3	67	1	21	4	3					000	0 3	32	4	0	0	0	0	4
4	4	49	1	15	5	3					000	0 4	42	3	0	0	0	0	4
5	5	69	2	15	6	3					000	0 5	52	2	0	0	0	0	4
6	5	69	1	15	6	3					000	0 6	54	1	0	0	0	0	4
7	1	12	0	12	2	0					0 7	000	11	0	0	0	0	0	0
8	3	67	0	21	4	3					12	000	36	0	0	1	0	0	0
							B		33	21									
9	6	2	0	2	3	2					34	000	61	0	1	2	0	0	0
							D	34		71									
10	4	49	0	15	5	3					34	34	46	1	1	2	1	0	0
							B		43	91									
							B		44	92									
							N	000	45	82									
11	7	10	1	10	3	0					000	34	71	0	3	2	1	0	5
12	8	7	1	7	3	0					45	000	82	0	3	3	0	0	1
13	9	7	1	7	3	0					44	000	92	0	2	4	0	0	1
14	9	7	0	7	3	0					43	000	91	0	1	5	0	0	0
15	7	10	0	10	3	0					93	000	72	0	1	6	0	0	0
16	5	69	0	15	6	3					73	71	51	0	1	6	1	1	0
							B		53	22									
17	8	7	0	7	3	0					56	000	81	0	2	7	0	0	0
18	3	67	4	21	4	3					83	000	35	0	2	8	0	0	1
19	2	4	1	4	0	0					53	000	22	0	1	9	0	0	2
20	2	4	0	4	0	0					33	000	21	0	0	10	0	0	2

number of links in a branch; and LB-label used in generating the Execution Sequence.

Execution sequence—The Operational Sequence contains much more information than is necessary to generate computer instructions but serves well as documentation and conformation of correct transformation. A set of dynamic macro instructions, called the Execution Sequence, is created to reduce the processing load in the subsequent Translation operations. The Execution Sequence for the Hit Sorting Program is given in Table VIII. A macro is generated when the Input Index (Q) is zero or the value of LB is either 1 or 2 in the Operational Sequence. The macro consists of the element label, LORT, LSRT, LIRT and the specific number of any "prepare." For example the Branch on Compare produces the macro in line 4, whose symbols BC03403404601 stand for:

element label = BC,

LORT = 034,

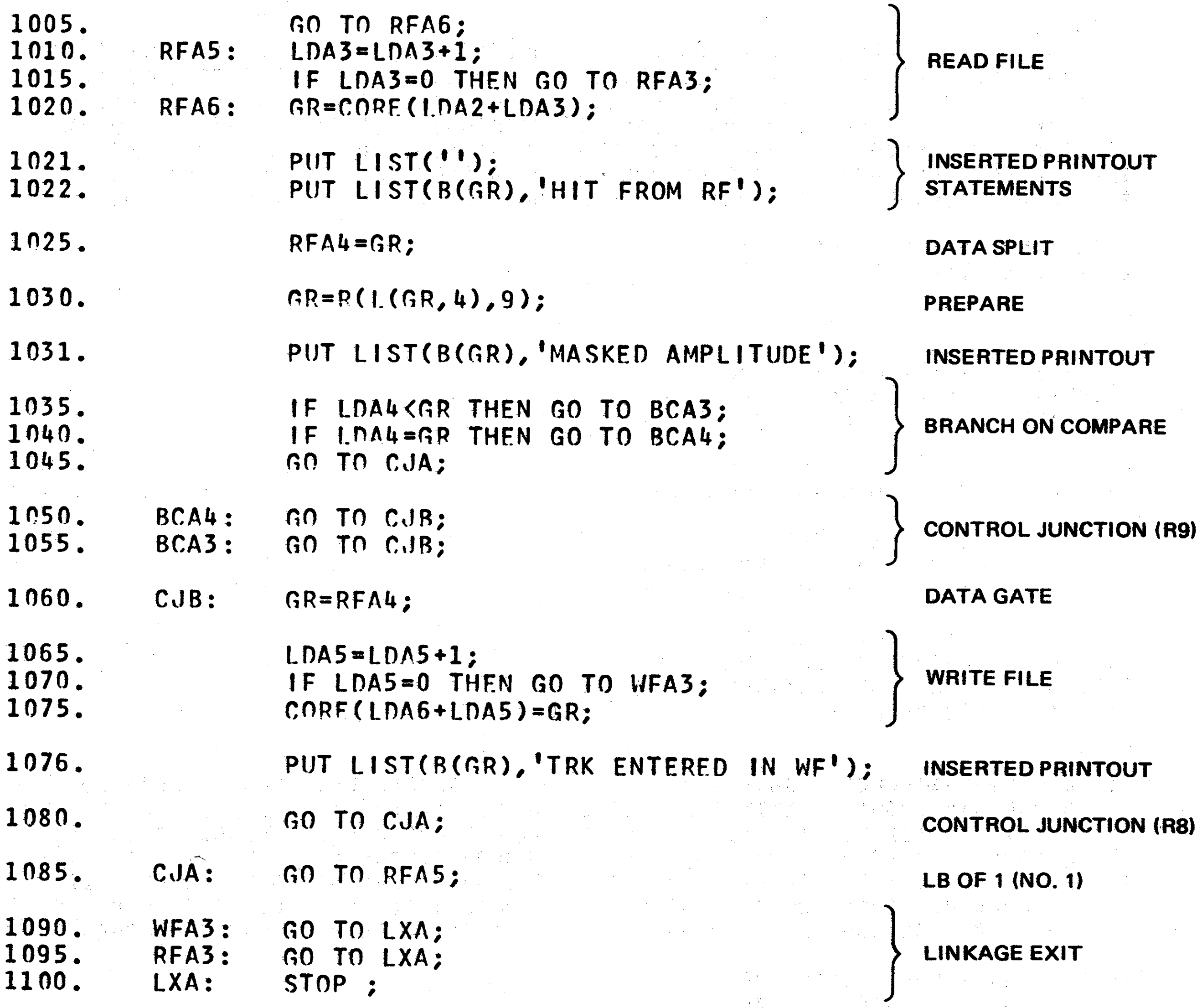
LSRT = 034,

LIRT = 046,

and the number of the prepare operation = 01.

Translation

a. Simulation—For the purpose of checking out the circuit on an interactive terminal each dynamic macro



```
1005.               GO TO RFA6;                                  }
1010.    RFA5:      LDA3=LDA3+1;                                 } READ FILE
1015.               IF LDA3=0 THEN GO TO RFA3;                   }
1020.    RFA6:      GR=CORE(LDA2+LDA3);                          }

1021.               PUT LIST('');                                } INSERTED PRINTOUT
1022.               PUT LIST(B(GR),'HIT FROM RF');               } STATEMENTS

1025.               RFA4=GR;                                       DATA SPLIT

1030.               GR=R(L(GR,4),9);                               PREPARE

1031.               PUT LIST(B(GR),'MASKED AMPLITUDE');            INSERTED PRINTOUT

1035.               IF LDA4<GR THEN GO TO BCA3;                  }
1040.               IF LDA4=GR THEN GO TO BCA4;                  } BRANCH ON COMPARE
1045.               GO TO CJA;                                   }

1050.    BCA4:      GO TO CJB;                                   } CONTROL JUNCTION (R9)
1055.    BCA3:      GO TO CJB;                                   }

1060.    CJB:       GR=RFA4;                                       DATA GATE

1065.               LDA5=LDA5+1;                                 }
1070.               IF LDA5=0 THEN GO TO WFA3;                   } WRITE FILE
1075.               CORE(LDA6+LDA5)=GR;                          }

1076.               PUT LIST(B(GR),'TRK ENTERED IN WF');           INSERTED PRINTOUT

1080.               GO TO CJA;                                     CONTROL JUNCTION (R8)

1085.    CJA:       GO TO RFA5;                                    LB OF 1 (NO. 1)

1090.    WFA3:      GO TO LXA;                                   }
1095.    RFA3:      GO TO LXA;                                   } LINKAGE EXIT
1100.    LXA:       STOP ;                                       }
```

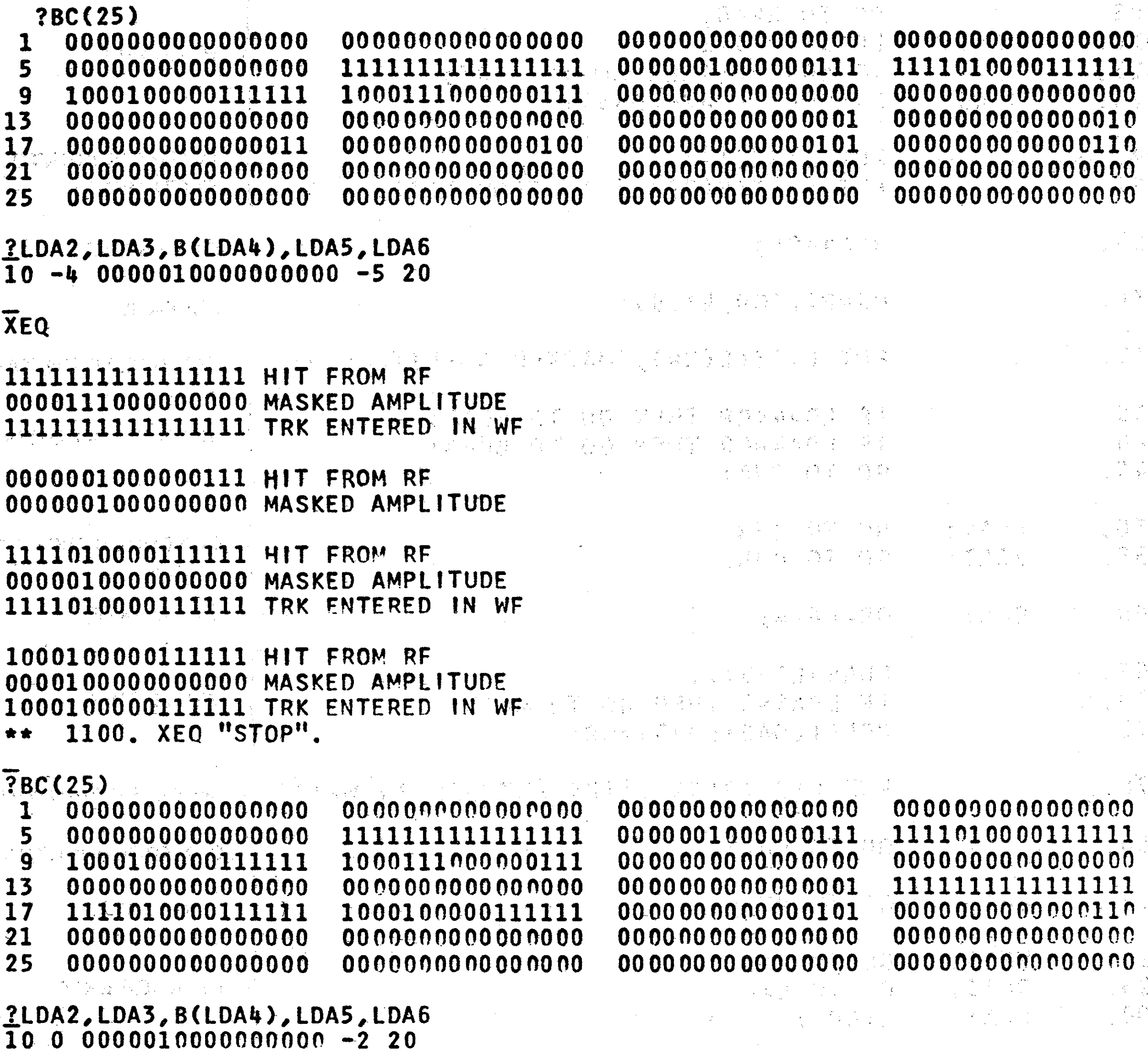
Figure 4—CPS code

was expanded to Conversational Programming System statements which are a subset of PL/I statements. The CPS statements for the Hit Sorting Program generated by macro expansion techniques are given in Figure 4. These statements are numbered in fives; all other statements have been added to provide simulation output. A description of the origin of each set of statements is added at the right. The reference numbers (R) have been replaced during the expansion by element labels (plus a unique letter) to help make the mnemonics more meaningful. An array called CORE is used as the file accessed by the Read File and Write File elements. The Linkage Data (LD) inputs to the circuit must be initialized or else they will be undefined.

The actual simulation output in Figure 5 illustrates a typical run. CORE and Linkage Data inputs are assumed to be previously entered. The binary representation of CORE and the decimal or binary values of the LD inputs are at the top of the figure (i.e., ? BC(30) and ? LDA2, LDA3 etc.). The same type of simulation printout is at the botton of the figure and represents the results of executing the simulation.

When XEQ is entered at the terminal the CPS code is executed as in the center section of Figure 5. The resulting information is generated by the PUT LIST statements inserted in the CPS code. The STOP represents exiting the circuit via the Linkage Exit element.

The actual data used in the simulation are four hits (i.e., LDA2 = N = −4) with the amplitudes of 7, 1, 2, and 4, respectively. The threshold (T = LDA4) has



```
  ?BC(25)
 1  0000000000000000  0000000000000000  0000000000000000  0000000000000000
 5  0000000000000000  1111111111111111  0000001000000111  1111010000111111
 9  1000100000111111  1000111000000111  0000000000000000  0000000000000000
13  0000000000000000  0000000000000000  0000000000000001  0000000000000010
17  0000000000000011  0000000000000100  0000000000000101  0000000000000110
21  0000000000000000  0000000000000000  0000000000000000  0000000000000000
25  0000000000000000  0000000000000000  0000000000000000  0000000000000000

?LDA2,LDA3,B(LDA4),LDA5,LDA6
10 -4 0000010000000000 -5 20

XEQ

1111111111111111 HIT FROM RF
0000111000000000 MASKED AMPLITUDE
1111111111111111 TRK ENTERED IN WF

0000001000000111 HIT FROM RF
0000001000000000 MASKED AMPLITUDE

1111010000111111 HIT FROM RF
0000010000000000 MASKED AMPLITUDE
1111010000111111 TRK ENTERED IN WF

1000100000111111 HIT FROM RF
0000100000000000 MASKED AMPLITUDE
1000100000111111 TRK ENTERED IN WF
**  1100. XEQ "STOP".

?BC(25)
 1  0000000000000000  0000000000000000  0000000000000000  0000000000000000
 5  0000000000000000  1111111111111111  0000001000000111  1111010000111111
 9  1000100000111111  1000111000000111  0000000000000000  0000000000000000
13  0000000000000000  0000000000000000  0000000000000001  1111111111111111
17  1111010000111111  1000100000111111  0000000000000101  0000000000000110
21  0000000000000000  0000000000000000  0000000000000000  0000000000000000
25  0000000000000000  0000000000000000  0000000000000000  0000000000000000

?LDA2,LDA3,B(LDA4),LDA5,LDA6
10 0 0000010000000000 -2 20
```

Figure 5—CPS simulation output

been set to 2 in the appropriate bit field. Thus, the first, third, and fourth hits are greater than or equal to the threshold amplitude. Those hits have been entered as tracks in the TRK file and are seen in CORE as the 16th, 17th, and 18th (i.e., LDA6−LDA5=16, 17, 18) entries. Upon completion, the HIT index (LDA3) has been decremented to zero which caused an exit from the Read File.

b. DAP Code Generation—The Honeywell DDP-516 is the target computer on which the example circuit must execute. The DAP assembly language for this computer is easily understood from the instruction mnemonics once it is recognized that the computer has a single accumulator and a single index register. The DAP code in Figure 6 was produced in the same manner as was the CPS code. Again, comments have been placed beside each set of assembly language instructions which represent one element. The circuit was constructed as a subroutine to retain modularity and compatibility with other system components.

INTEGRATION AND TESTING OF COMPLEX PROGRAMS

A large-scale data processing program can be broken down into a number of Data Flow Circuits. These can then be assembled into a block diagram in which each block is an individual Data Flow Circuit. Each representation block can be considered as a special "Macro" element, connected to other blocks by data and control signal inputs and outputs, just as are elements in a Data Flow Circuit. The integration of Data Flow Circuits can be accomplished by the use of an interactive terminal and a special Integration Program in a manner similar to that used in constructing Data Flow Circuits. This process is facilitated by linkage routing elements described earlier. The Integration program therefore serves the purpose of a "Linkage Editor".

An example of such a Data Flow diagram is shown in Figure 7. This diagram represents the Track Prediction module of the Automatic Tracking Program for a surveillance radar. The blocks are represented by rectangles and the data files by squares. The Track Coordinate Computation Circuit is seen to be near the center of the diagram.

In like manner the Track Prediction module itself can be considered a block in a higher level diagram representing the entire data processing program. In this way an orderly and flexible format for the total program can be obtained.

Program Dictionary—The efficient design of a complex program requires careful definition, organization, and maintenance of all terms used in the program. The list of these characteristics has to be assembled during the design process of circuit design. When complete it provides a basis for complete documentation of the program and facilitates future changes.

```
ORG     '2000       Entered as Origin

DAC     **          Linkage Entry

JMP     RFA6    }
IRS     LDA3    }
SKP             }   Read File (HIT)
JMP*    HSP     }   and one terminal of
LDX     LDA3    }   Linkage Exit
LDA     LDA2, 1 }

STA     RFA4        Data Split

ANA     = '7000     Prepare on Branch on Compare

CAS     LDA4    }
JMP     RFA5    }   Branch on Compare
JMP     BCA4    }

LDA     RFA4        Data Gate

IRS     LDA5    }
SKP             }   Write File (TRK)
JMP*    HSP     }   and other terminal of
LDX     LDA5    }   Linkage Exit
STA     LDA6, 1 }
JMP     RFA5    }

EQU     '500        Index to Hit File = LOC 500
DAC     '7000       Base to Hit File 7000
BSZ     1           Temp. Store for DS
OCT     '2000       Threshold Amplitude of 2
EQU     '400        Index to TRK File = LOC 400
DAC     '6000       Base to Hit File 6000
END
```

Figure 6—DAP code

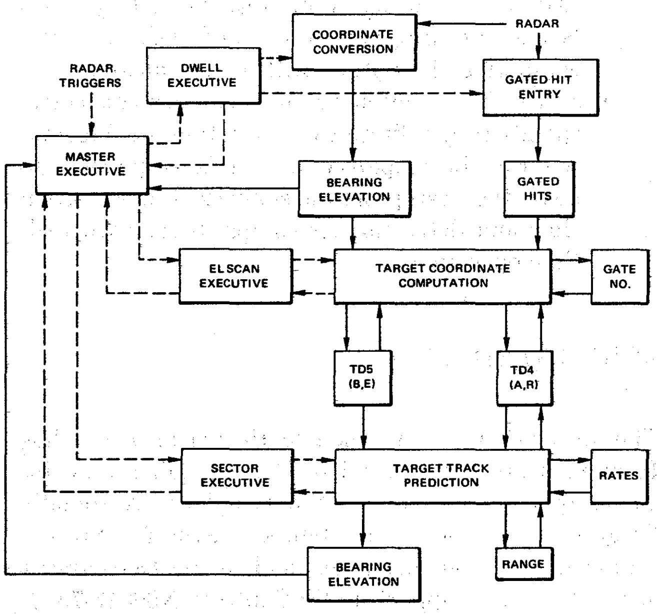


Figure 7—Track prediction module

The data required for this list include the code names, definition, format, constituent parts, and cross-reference of all variables, constants, data files, circuit blocks and program modules.

Once the above terms have been defined, their subsequent manipulation requires only reference by code name. This is expected to save a great deal of housekeeping, and to eliminate a major source of error.

Program checkout—The most laborious and time consuming part of programming is the "debugging" phase. Graphic Automatic Programming facilitates the production of a correct program in the following ways:

1. The pattern of data and logic flow is made highly visible and hence minimizes errors at the source.
2. Any inconsistencies in the circuit are detected by the Transformation program by checking the Element Interconnection Matrix and through rules regarding allowed element linkages.
3. The graphics terminal enables the designer to correct errors on-line and to verify that they have been eliminated.
4. The designer can test a circuit on the terminal by entering sample inputs and reading out the

resulting outputs. He can also design another Data Circuit which would test the first by simulating the program input and automatically comparing the output with requirements.

5. Since the functioning of each element corresponds to a definite execution time in the computer to be employed, it is readily possible to have the test program simulate the execution time and determine its compatibility with real-time operation.

ACKNOWLEDGMENT

The authors wish to thank Lee Hoevel of The Johns Hopkins University, Applied Physics Laboratory for his early contributions to the Graphical Automatic Programming concept, to James Austin for his encouragement of our activities, and to the Department of the Navy for support under Contract N00017-72-C-4401.

The initial concept of Graphical Automatic Programming was described in the September-October 1969 issue of the *APL Technical Digest* (The Johns Hopkins University, Applied Physics Laboratory).

APPENDIX

The "Target Coordinate Computation Circuit" in Figure A1 is a more sophisticated example of the type of process intended for GAP. The interconnection Matrix is given as Table AI and the Execution Sequence in Table AII. The CPS expanded statements and simulation printout are in Figures A2 and A3. The DAP code is given in Figure A4.

The logic of the "Target Coordinate Computation Circuit" operates as follows:

1. If no prior hit exists in the target data file (TD4), set the number of hits to 1 and store coordinates of new hit in the target data files (TD4, TD5).
2. If a previous hit exists, but does not coincide in range with the new hit from the gated hit file (GH), exit to multiple target routine.
3. If the previous hit correlates in range, increment number of hits and store the target coordinates of strongest hit.

The circuit is activated by a control signal at Read File, element R4.

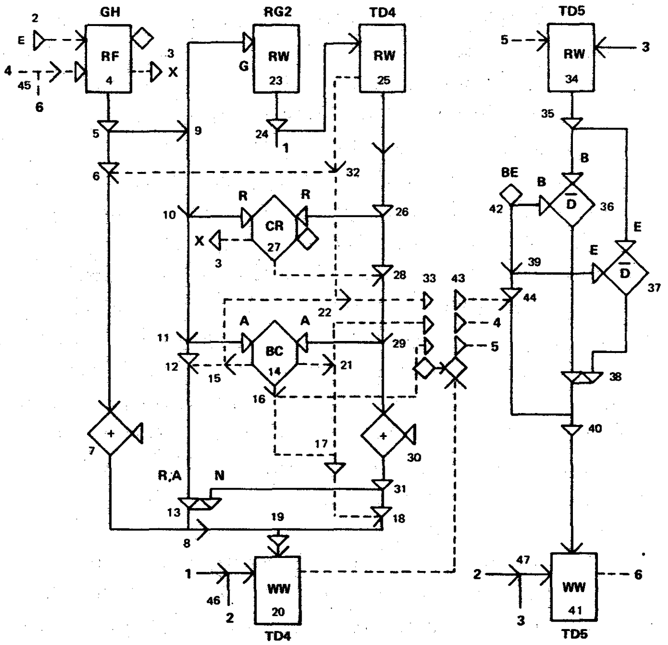


Figure A1—Target coordination computation circuit

The RF element reads out a word (i.e., a gated hit) consisting of the Gate number (G), Range (R), and Amplitude (A).

The Gate number is extracted and used to address the Read Word element R23, which extracts the track number from the Range Gate 2 file (RG2).

The track number is used to address the Read Word element R25, which extracts the data contained in the Track Data 4 file (TD4).

If the TD4 file is empty, a control signal is sent to the Data Gate element R6.

The output of element R6 is incremented by 1 in the Add element R7 and stored in TD4 by the Write Word element R20.

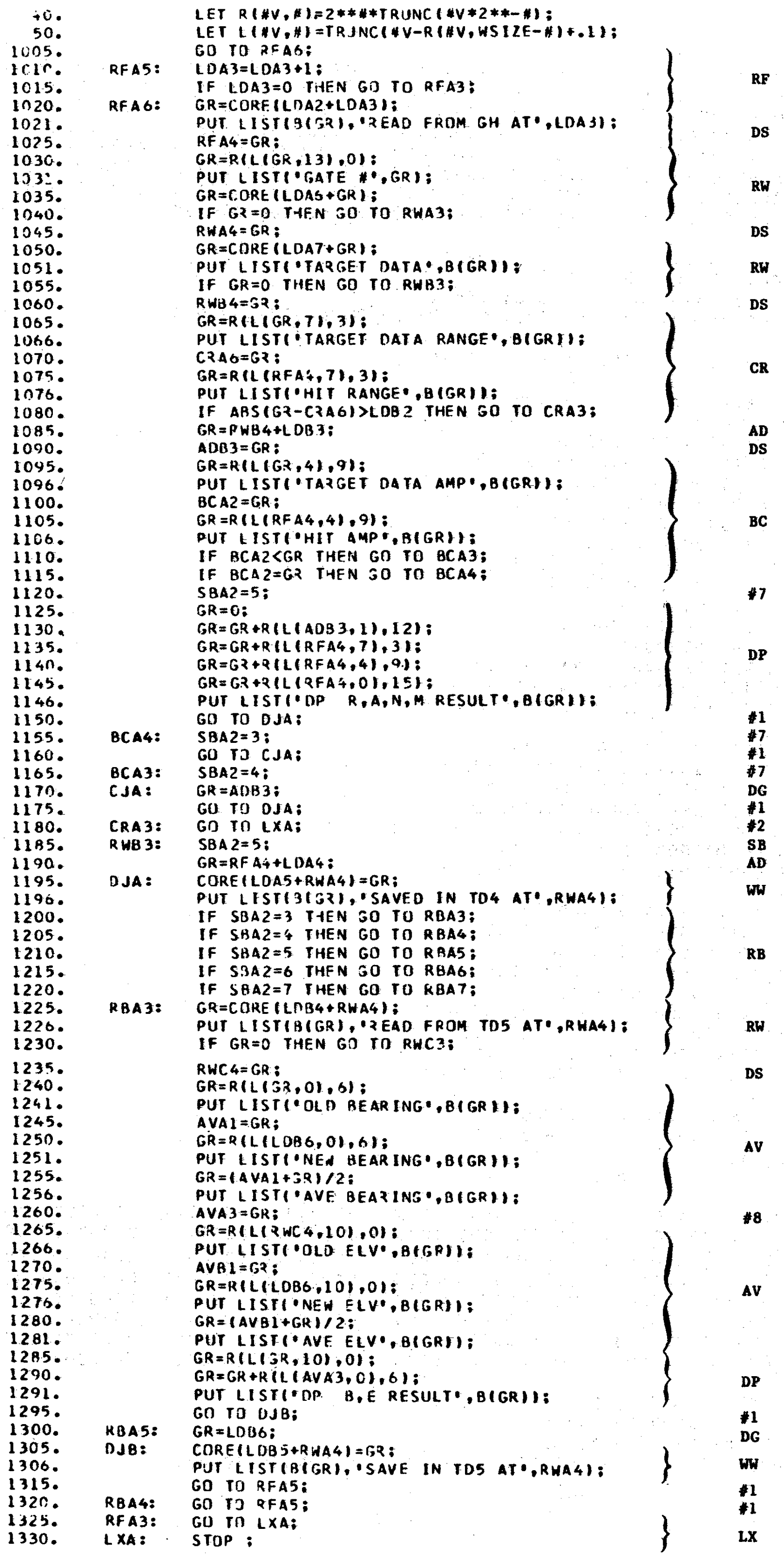
If the TD4 file is not empty, the word (Range, Amplitude, Number of hits) is extracted and sent to the Correlate element R27.

If the range of the gated hit does not correlate with the range stored previously, a control signal is emitted to activate another program dealing with multiple tracks.

If the ranges correlate, the amplitudes are compared in the Branch on Compare element R14.

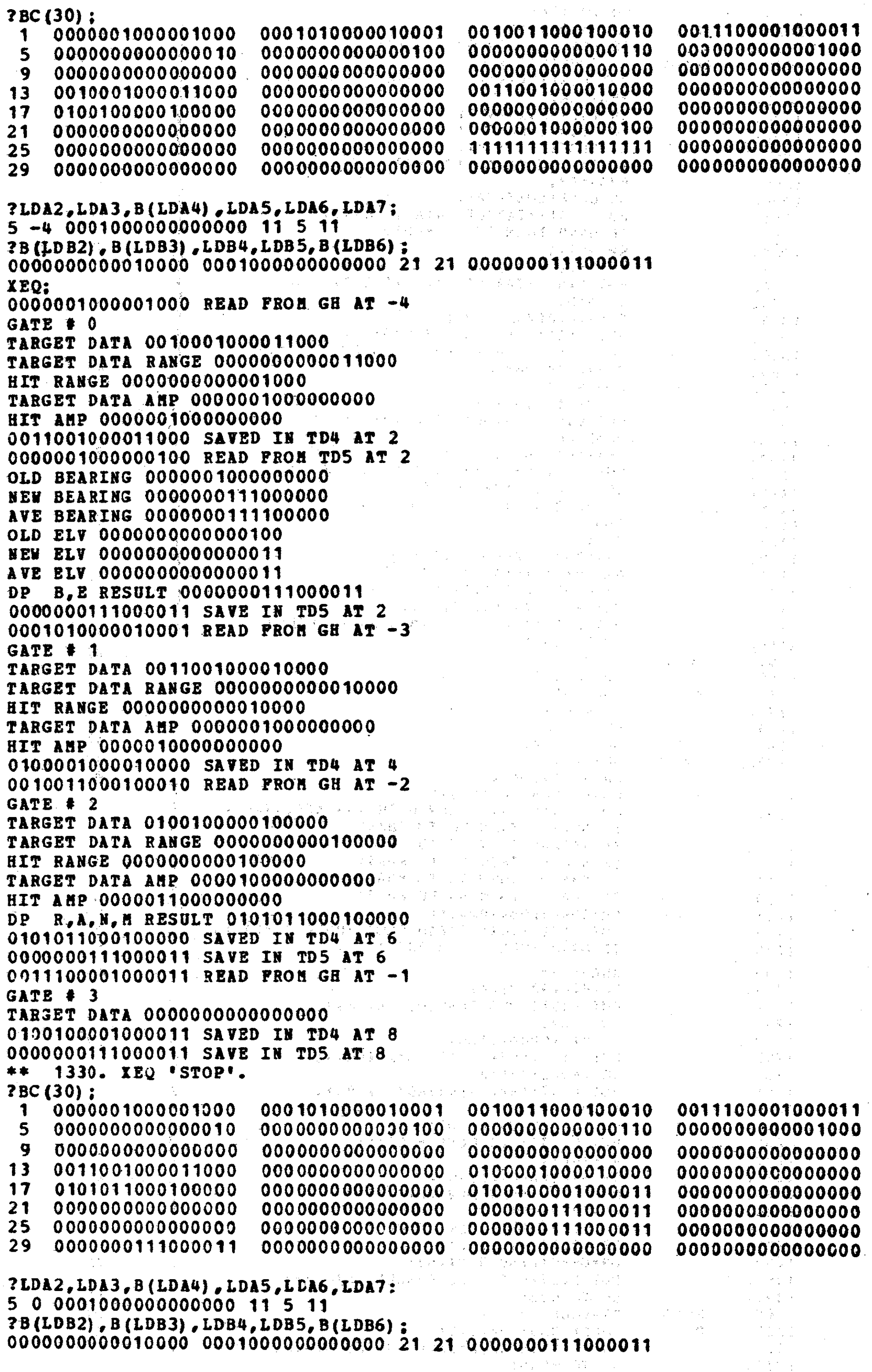
Also the number of hits is incremented by element R30.

If the amplitude of the gated hit is greater than that of the old hit from TD4, its range and amplitude are combined with the incremented number of hits, N, and stored in TD4.



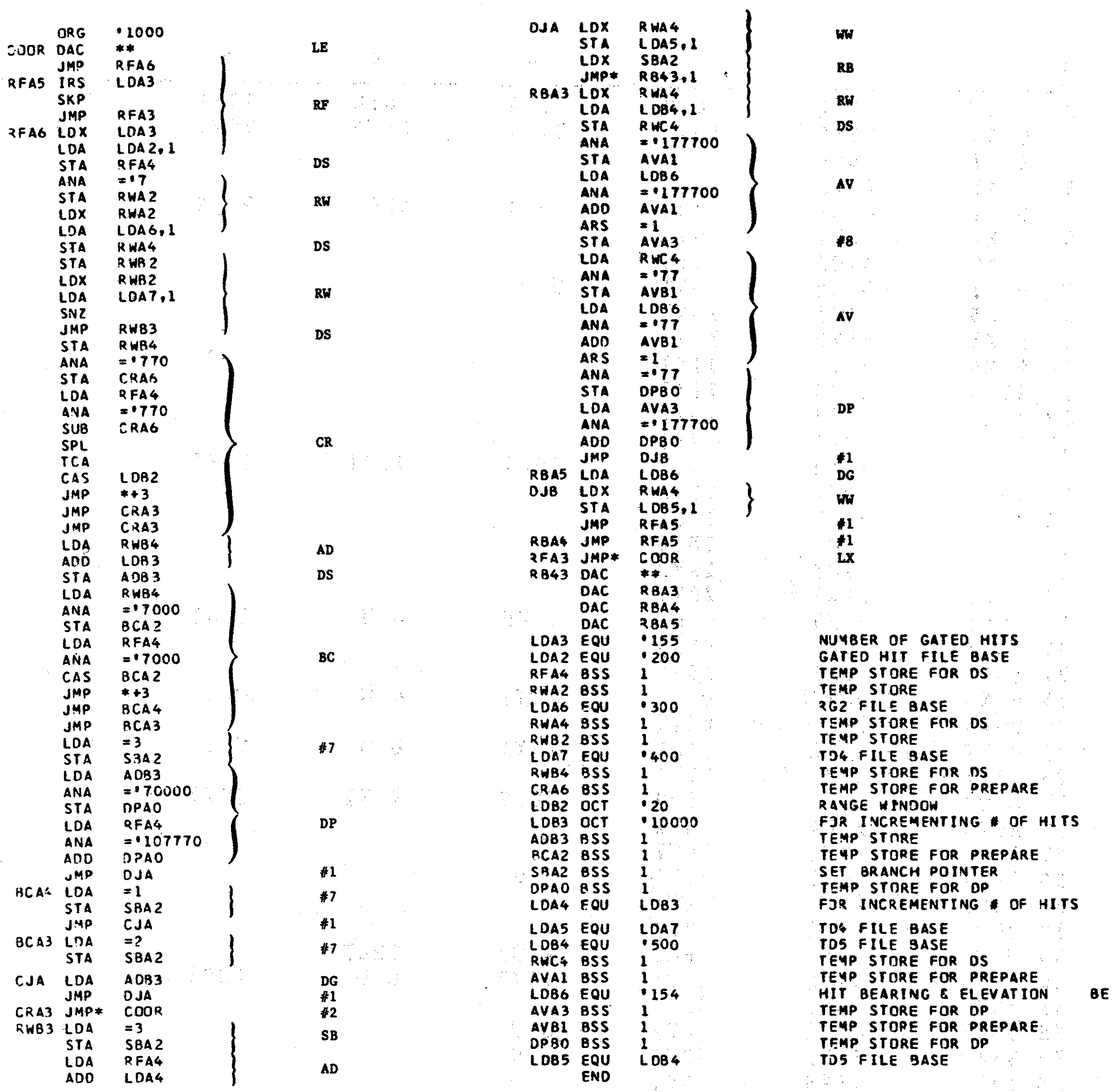
```
  40.           LET R(#V,#)=2**#*TRUNC(#V*2**-#);
  50.           LET L(#V,#)=TRJNC(#V-R(#V,WSIZE-#)+.1);
1005.           GO TO RFA6;
1010.  RFA5:    LDA3=LDA3+1;                                          RF
1015.           IF LDA3=0 THEN GO TO RFA3;
1020.  RFA6:    GR=CORE(LDA2+LDA3);
1021.           PUT LIST(B(GR),'READ FROM GH AT',LDA3);               DS
1025.           RFA4=GR;
1030.           GR=R(L(GR,13),0);
1031.           PUT LIST('GATE #',GR);                                RW
1035.           GR=CORE(LDA6+GR);
1040.           IF GR=0 THEN GO TO RWA3;
1045.           RWA4=GR;                                              DS
1050.           GR=CORE(LDA7+GR);
1051.           PUT LIST('TARGET DATA',B(GR));                        RW
1055.           IF GR=0 THEN GO TO RWB3;
1060.           RWB4=GR;                                              DS
1065.           GR=R(L(GR,7),3);
1066.           PUT LIST('TARGET DATA RANGE',B(GR));
1070.           CRA6=GR;                                              CR
1075.           GR=R(L(RFA4,7),3);
1076.           PUT LIST('HIT RANGE',B(GR));
1080.           IF ABS(GR-CRA6)>LDB2 THEN GO TO CRA3;
1085.           GR=PWB4+LDB3;                                         AD
1090.           ADB3=GR;                                              DS
1095.           GR=R(L(GR,4),9);
1096.           PUT LIST('TARGET DATA AMP',B(GR));
1100.           BCA2=GR;
1105.           GR=R(L(RFA4,4),9);                                    BC
1106.           PUT LIST('HIT AMP',B(GR));
1110.           IF BCA2<GR THEN GO TO BCA3;
1115.           IF BCA2=GR THEN GO TO BCA4;
1120.           SBA2=5;                                               #7
1125.           GR=0;
1130.           GR=GR+R(L(ADB3,1),12);
1135.           GR=GR+R(L(RFA4,7),3);                                 DP
1140.           GR=GR+R(L(RFA4,4),9);
1145.           GR=GR+R(L(RFA4,0),15);
1146.           PUT LIST('DP  R,A,N,M RESULT',B(GR));
1150.           GO TO DJA;                                            #1
1155.  BCA4:    SBA2=3;                                               #7
1160.           GO TO CJA;                                            #1
1165.  BCA3:    SBA2=4;                                               #7
1170.  CJA:     GR=ADB3;                                              DG
1175.           GO TO DJA;                                            #1
1180.  CRA3:    GO TO LXA;                                            #2
1185.  RWB3:    SBA2=5;                                               SB
1190.           GR=RFA4+LDA4;                                         AD
1195.  DJA:     CORE(LDA5+RWA4)=GR;                                   WW
1196.           PUT LIST(B(GR),'SAVED IN TD4 AT',RWA4);
1200.           IF SBA2=3 THEN GO TO RBA3;
1205.           IF SBA2=4 THEN GO TO RBA4;
1210.           IF SBA2=5 THEN GO TO RBA5;                            RB
1215.           IF SBA2=6 THEN GO TO RBA6;
1220.           IF SBA2=7 THEN GO TO RBA7;
1225.  RBA3:    GR=CORE(LDB4+RWA4);
1226.           PUT LIST(B(GR),'READ FROM TD5 AT',RWA4);              RW
1230.           IF GR=0 THEN GO TO RWC3;
1235.           RWC4=GR;                                              DS
1240.           GR=R(L(GR,0),6);
1241.           PUT LIST('OLD BEARING',B(GR));
1245.           AVA1=GR;
1250.           GR=R(L(LDB6,0),6);                                    AV
1251.           PUT LIST('NEW BEARING',B(GR));
1255.           GR=(AVA1+GR)/2;
1256.           PUT LIST('AVE BEARING',B(GR));
1260.           AVA3=GR;                                              #8
1265.           GR=R(L(RWC4,10),0);
1266.           PUT LIST('OLD ELV',B(GR));
1270.           AVB1=GR;
1275.           GR=R(L(LDB6,10),0);                                   AV
1276.           PUT LIST('NEW ELV',B(GR));
1280.           GR=(AVB1+GR)/2;
1281.           PUT LIST('AVE ELV',B(GR));
1285.           GR=R(L(GR,10),0);
1290.           GR=GR+R(L(AVA3,0),6);                                 DP
1291.           PUT LIST('DP  B,E RESULT',B(GR));
1295.           GO TO DJB;                                            #1
1300.  RBA5:    GR=LDB6;                                              DG
1305.  DJB:     CORE(LDB5+RWA4)=GR;                                   WW
1306.           PUT LIST(B(GR),'SAVE IN TD5 AT',RWA4);
1315.           GO TO RFA5;                                           #1
1320.  RBA4:    GO TO RFA5;                                           #1
1325.  RFA3:    GO TO LXA;
1330.  LXA:     STOP ;                                                LX
```

Figure A2—CPS code



```
?BC(30);
 1  0000001000001000  0001010000010001  0010011000100010  0011100001000011
 5  0000000000000010  0000000000000100  0000000000000110  0000000000001000
 9  0000000000000000  0000000000000000  0000000000000000  0000000000000000
13  0010001000011000  0000000000000000  0011001000010000  0000000000000000
17  0100100000100000  0000000000000000  0000000000000000  0000000000000000
21  0000000000000000  0000000000000000  0000001000000100  0000000000000000
25  0000000000000000  0000000000000000  1111111111111111  0000000000000000
29  0000000000000000  0000000000000000  0000000000000000  0000000000000000

?LDA2,LDA3,B(LDA4),LDA5,LDA6,LDA7;
5 -4 0001000000000000 11 5 11
?B(LDB2),B(LDB3),LDB4,LDB5,B(LDB6);
0000000000010000 0001000000000000 21 21 0000000111000011
XEQ;
0000001000001000 READ FROM GH AT -4
GATE # 0
TARGET DATA 0010001000011000
TARGET DATA RANGE 0000000000011000
HIT RANGE 0000000000001000
TARGET DATA AMP 0000001000000000
HIT AMP 0000001000000000
0011001000011000 SAVED IN TD4 AT 2
0000001000000100 READ FROM TD5 AT 2
OLD BEARING 0000001000000000
NEW BEARING 0000000111000000
AVE BEARING 0000000111100000
OLD ELV 0000000000000100
NEW ELV 0000000000000011
AVE ELV 0000000000000011
DP  B,E RESULT 0000000111000011
0000000111000011 SAVE IN TD5 AT 2
0001010000010001 READ FROM GH AT -3
GATE # 1
TARGET DATA 0011001000010000
TARGET DATA RANGE 0000000000010000
HIT RANGE 0000000000010000
TARGET DATA AMP 0000001000000000
HIT AMP 0000010000000000
0100001000010000 SAVED IN TD4 AT 4
0010011000100010 READ FROM GH AT -2
GATE # 2
TARGET DATA 0100100000100000
TARGET DATA RANGE 0000000000100000
HIT RANGE 0000000000100000
TARGET DATA AMP 0000100000000000
HIT AMP 0000011000000000
DP  R,A,N,M RESULT 0101011000100000
0101011000100000 SAVED IN TD4 AT 6
0000000111000011 SAVE IN TD5 AT 6
0011100001000011 READ FROM GH AT -1
GATE # 3
TARGET DATA 0000000000000000
0100100001000011 SAVED IN TD4 AT 8
0000000111000011 SAVE IN TD5 AT 8
**  1330. XEQ 'STOP'.
?BC(30);
 1  0000001000001000  0001010000010001  0010011000100010  0011100001000011
 5  0000000000000010  0000000000000100  0000000000000110  0000000000001000
 9  0000000000000000  0000000000000000  0000000000000000  0000000000000000
13  0011001000011000  0000000000000000  0100001000010000  0000000000000000
17  0101011000100000  0000000000000000  0100100001000011  0000000000000000
21  0000000000000000  0000000000000000  0000000111000011  0000000000000000
25  0000000000000000  0000000000000000  0000000111000011  0000000000000000
29  0000000111000011  0000000000000000  0000000000000000  0000000000000000

?LDA2,LDA3,B(LDA4),LDA5,LDA6,LDA7:
5 0 0001000000000000 11 5 11
?B(LDB2),B(LDB3),LDB4,LDB5,B(LDB6);
0000000000010000 0001000000000000 21 21 0000000111000011
```

Figure A3—CPS simulation output



```
      ORG   '1000
COOR  DAC   **                  LE
      JMP   RFA6
RFA5  IRS   LDA3
      SKP                       RF
      JMP   RFA3
RFA6  LDX   LDA3
      LDA   LDA2,1
      STA   RFA4                DS
      ANA   ='7
      STA   RWA2                RW
      LDX   RWA2
      LDA   LDA6,1
      STA   RWA4                DS
      STA   RWB2
      LDX   RWB2
      LDA   LDA7,1              RW
      SNZ
      JMP   RWB3                DS
      STA   RWB4
      ANA   ='770
      STA   CRA6
      LDA   RFA4
      ANA   ='770
      SUB   CRA6
      SPL                       CR
      TCA
      CAS   LDB2
      JMP   *+3
      JMP   CRA3
      JMP   CRA3
      LDA   RWB4                AD
      ADD   LDB3
      STA   ADB3                DS
      LDA   RWB4
      ANA   ='7000
      STA   BCA2
      LDA   RFA4
      ANA   ='7000              BC
      CAS   BCA2
      JMP   *+3
      JMP   BCA4
      JMP   BCA3
      LDA   =3                  #7
      STA   SBA2
      LDA   ADB3
      ANA   ='70000
      STA   DPAO
      LDA   RFA4                DP
      ANA   ='107770
      ADD   DPAO
      JMP   DJA                 #1
BCA4  LDA   =1                  #7
      STA   SBA2
      JMP   CJA                 #1
BCA3  LDA   =2                  #7
      STA   SBA2
CJA   LDA   ADB3                DG
      JMP   DJA                 #1
CRA3  JMP*  COOR                #2
RWB3  LDA   =3                  SB
      STA   SBA2
      LDA   RFA4                AD
      ADD   LDA4
```

```
DJA   LDX   RWA4
      STA   LDA5,1              WW
      LDX   SBA2                RB
      JMP*  RB43,1
RBA3  LDX   RWA4
      LDA   LDB4,1              RW
      STA   RWC4                DS
      ANA   ='177700
      STA   AVA1
      LDA   LDB6                AV
      ANA   ='177700
      ADD   AVA1
      ARS   =1
      STA   AVA3                #8
      LDA   RWC4
      ANA   ='77
      STA   AVB1
      LDA   LDB6                AV
      ANA   ='77
      ADD   AVB1
      ARS   =1
      ANA   ='77
      STA   DPBO
      LDA   AVA3                DP
      ANA   ='177700
      ADD   DPBO
      JMP   DJ8                 #1
RBA5  LDA   LDB6                DG
DJB   LDX   RWA4                WW
      STA   LDB5,1
      JMP   RFA5                #1
RBA4  JMP   RFA5                #1
RFA3  JMP*  COOR                LX
RB43  DAC   **
      DAC   RBA3
      DAC   RBA4
      DAC   RBA5
LDA3  EQU   '155                NUMBER OF GATED HITS
LDA2  EQU   '200                GATED HIT FILE BASE
RFA4  BSS   1                   TEMP STORE FOR DS
RWA2  BSS   1                   TEMP STORE
LDA6  EQU   '300                RG2 FILE BASE
RWA4  BSS   1                   TEMP STORE FOR DS
RWB2  BSS   1                   TEMP STORE
LDA7  EQU   '400                TD4 FILE BASE
RWB4  BSS   1                   TEMP STORE FOR DS
CRA6  BSS   1                   TEMP STORE FOR PREPARE
LDB2  OCT   '20                 RANGE WINDOW
LDB3  OCT   '10000              FOR INCREMENTING # OF HITS
ADB3  BSS   1                   TEMP STORE
BCA2  BSS   1                   TEMP STORE FOR PREPARE
SBA2  BSS   1                   SET BRANCH POINTER
DPAO  BSS   1                   TEMP STORE FOR DP
LDA4  EQU   LDB3                FOR INCREMENTING # OF HITS
LDA5  EQU   LDA7                TD4 FILE BASE
LDB4  EQU   '500                TD5 FILE BASE
RWC4  BSS   1                   TEMP STORE FOR DS
AVA1  BSS   1                   TEMP STORE FOR PREPARE
LDB6  EQU   '154                HIT BEARING & ELEVATION    BE
AVA3  BSS   1                   TEMP STORE FOR DP
AVB1  BSS   1                   TEMP STORE FOR PREPARE
DPBO  BSS   1                   TEMP STORE FOR DP
LDB5  EQU   LDB4                TD5 FILE BASE
      END
```

Figure A4—DAP code

If the amplitude previously stored is greater than or equal to that of the gated hit, the previous values with N incremented are replaced in TD4.

After the WW element R20 functions, the resulting control signal actuates the Read Branch element R43, which emits a control signal from whichever terminal corresponds to the control input stored in the Set Branch element R33. The SB element was activated as a result of the operation of the BC element.

If the gated hit amplitude had been greater, the bearing and elevation (BE) of the new hit are stored in Track Data 5 file (TD5) upon exiting SB at its upper connection.

If the amplitudes had been equal, the bearing and elevation previously stored in the TD5 file are extracted, averaged with those of the new hit by elements R36 and R37, packed into a single word by R38, and replaced in RD5.

If the amplitude of the previously stored hit had been greater or after B and E had been stored in RD5, a control signal reactivates RF element R4 to read out the next gated hit.

When all gated hits in the GH file have been processed a control signal exits the circuit to the next program.

```
LD008000011
PS000016421
LE017000021
RF022000046
DS044000051
RW00004423209
DS234000241
RW000234252
DS254000261
CR0002542760304
DG274000282
AD283281301
DS303000311
BC0002811420102
CS145000151
#7145000335
DG145000122
DP123121132
#1133000191
CS144000161
#7144000333
#1144000171
CS143000211
#7143000334
CJ143000172
DG173000182
#1183181192
#2273000032
CS253000321
SB253000335
#8332332432
DG253000062
AD063061071
DJ073000191
WW193000201
RB206000431
RW433000346
DS344000351
AV3443443610506
#8363363382
AV0003443710708
DP373000381
#1383000401
DG435000442
DJ443441402
WW403000411
#1416000045
#1434000045
LX043000031
```

TABLE A2—Execution Sequence

R	1	2	3	4	5	6	7	8
0	C	4Y1	4Y2	7Y2	20Y4	23Y1	25Y1	1J1
1	0 C8	27Y5	30Y2	34Y1	41Y4	42Y1	2J1	
2	1 C7	4O6						
3	4 I3	27I3						
4	0 X2	0 X3	3J1	5Y1	45C3	2 C2		
5	4 X4	6Y1	9Y1					
6	5 X2	32C2	7Y1					
7	6 X3	0 X4	8Y1	U U				
8	7 X3	13X3	19Y1					
9	5 X3	23Y2	10Y1					
10	9 X3	27Y2	11Y1					
11	10X3	14Y6	12Y1					
12	X X	15C2	13Y2					
13	31X2	12X3	8Y2					
14	U U	29X2	21J1	16J1	15J1	11X2		
15	14C5	12J2	22J1					
16	14C4	33J3	17J1					
17	16C3	21C2	18J2					
18	31X3	17C3	19Y2					
19	8 X3	18X3	20Y1					
20	19X3	46X3	U U	0 X5	U U	43J1		
21	14C3	17J2	33J4					
22	15C3	32C3	33J5					
23	0 X6	9 X2	U U	24Y1	U U	U U		
24	23X4	25Y2	46Y1					
25	0 X7	24X2	32J1	26Y1	U U	U U		
26	25X4	27Y6	28Y1					
27	U U	10X2	3J2	28J2	1 X2	26X2		
28	26X3	27C4	29Y1					
29	28X3	14Y2	30Y1					
30	29X3	1 X3	31Y1	U U				
31	30X3	13Y1	18Y1					
32	25C3	6J2	22J2					
33	U U	43Y2	16C2	21C3	22C3			
34	1 X4	47X2	U U	35Y1	U U	43C3		
35	34X4	37Y1	36Y1					
36	35X3	42X2	38Y2	U U				
37	35X2	39X2	38Y1	U U				
38	37X3	36X3	40Y1					
39	42X3	37Y2	44Y1					
40	38X3	44X3	41Y1					
41	40X3	47X3	U U	1 X5	U U	45J1		
42	1 X6	36Y2	39Y1					
43	20C6	33X2	34J6	45J2	44J2			
44	39X3	43C5	40Y2					
45	41C6	43C4	4J5					
46	24X3	47Y1	20Y2					
47	46X2	34Y2	41Y2					

TABLE A1—Interconnection Matrix

Systems for systems implementors—Some experiences from Bliss*

by WILLIAM A. WULF

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

The programming language Bliss was developed at Carnegie-Mellon University expressly for the purpose of writing software systems* and has been in use for over three years. A considerable number of systems have been written using it: compilers, interpreters, i/o systems, simulators, operating systems, etc. The language was designed and implemented in the conventional sense of an isolated language system, and relies on the file system, editors, debuggers, etc., provided by the manufacturer and/or other users. In this paper we shall not describe Bliss, that has been done elsewhere;[1,2] nor shall we attempt to justify the language design, that has also been done.[3,4] Rather, we shall attempt to analyze and evaluate the particular decision** to implement Bliss as an isolated language rather than as a piece of a more comprehensive system. Some comments are made on the implications of this analysis/evaluation on the shape that such a system might have.

A CHARACTERIZATION OF THE PROBLEM AREA

We shall restrict our discussion to the field of "systems programming." While there is no universally accepted definition of this term, it is useful to have some characterization of it against which to frame the discussion. In particular we can discern four properties of systems programs relevant to this discussion. They:

1. must be efficient on a particular machine;
2. are large, probably requiring several implementors;
3. are "real" in the sense that they are widely distributed and used frequently (perhaps continuously);
4. are rarely "finished," but rather are elements in a design/implementation feedback cycle.

These properties may be factored into two sets—technical issues (item 1), and program management issues, i.e., those that arise exclusively because the systems are large, real, and volatile (items 2, 3, and 4).

The technical issues relate primarily to efficiency of two types: local and global. In most cases software systems can at most tolerate moderate inefficiencies in their object code; in a few critical situations anything other than the most efficient possible machine code is unacceptable. Although the issue of efficiency is largely language/compiler related, it must be recognized that a more general statement applies: given the *logical* machine which a software implementation system (SIS) defines, and any discrepancy between that model and the hardware itself, that discrepancy may become critical in one of two ways: either the cumulative inefficiency due to the distributed effects of the discrepancy is significant, or the use of the construct which invokes the discrepancy produces intolerable inefficiencies in some local context. Although we cannot hope to design an SIS which eliminates the effect entirely, we must take some care with the conventions we adopt, both in the SIS itself and in the management tools given the user.

The managerial issues which arise in the construction of large, complex systems can be separated into two

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office for Scientific Research.

* Primarily for the PDP-10 although Bliss has now been implemented for several other machines.

** At the time, of course, the decision was made by default; the more ambitious alternative was not considered.

classes:

1. those which we presently believe to be solvable within the framework of a static, compilable language; and
2. those whose solution, at present, seems to require the construction of a "total" programming environment which, in addition to the language, includes editing, monitoring, etc.

Within these classes, the tools for program management come in three forms: those which help specify a global structure to a task (top down modularization), those which specify common elements of a fine structure (predictive bottom up modularization), and finally those related to the relatively mechanical aspects of file manipulation, editing, debugging, etc.

A VIEW OF THE PROBLEM

Most of the recent effort devoted to the design of languages and systems has been expended to improve the convenience with which a program may be written. While convenience is an important criterion, it should not be the only, or even the central, issue in the design of a system for implementing other systems. The notion that convenience in writing programs should be the central issue results from the naive view that software is simply designed and written. That view is fallacious in terms of the four properties listed above.

In particular, programming systems are never finished but are in a constant state of evolution. New features are added and old errors repaired. The more heavily a system is used, the more rapid the rate of evolution and repair. This situation seems inevitable so long as new application areas, all with slightly different requirements, continue to emerge. Thus, the central problem of devising a system for systems programming would appear to be that of providing mechanisms for enabling the programmer to cope with this evolution while satisfying technical constraints imposed by systems implementation in general.

The mechanisms by which programmers may cope with the evolution of a system are those which we have termed 'managerial' above. It is these mechanisms which are most prominently lacking in our current system implementation tools; the consequence of this lack is the introduction of peripheral modifications which subvert and distort the original structure of a system and lead to inefficient, "dirty" systems.

WHAT IS A "GOOD" MANAGEMENT TOOL?

If the central problem of systems programming is that of coping with the evolutionary nature of systems, then a good tool is one which creates an environment in which this is relatively easier to do. Moreover, given an existing system and the desire to modify it in some way, the difficulty of making that modification is directly related to the extent of its interaction with what already exists. Modifications whose effects are localized are easy to make. Modifications whose effects are global—whether due to a large number of textual or conceptual interactions—are difficult to make.

In general the "goodness" of a tool appears, then, to be directly related to the degree to which its use permits and encourages decoupling, isolating, decisions and hence localizing their effect. Thus, to pick two trite examples, subroutines and macros are good tools precisely because they permit isolation of a computational representation (a particular encoding) from the intended effect of that computation.

MANAGEMENT FUNCTIONS PROVIDED BY LANGUAGE

One view of the recent history of the development of programming systems holds that it has been a search for panaceas. According to this view the development of large 'shell' languages (e.g., PL/I), extensible languages, time-sharing, etc., have each in turn been sponsored, in part, because they promised to be *the* solution to providing more convenient, accessible, and cost/effective computing. Whether this view has complete validity or not, we do not want to fall into the trap of looking to the mystic word 'system' to remedy the ills of past software development projects. Therefore we will first discuss some of the management facilities which can and should be provided at the language level.

The decision to use any higher-level language represents a good program management decision to the extent that the structuring facilities of the language are used in the implementation. It represents a sound technical decision to the extent that they are usable. For example, Bliss chose to include Algol block-structure, scope and extent of variables, functions, boolean and arithmetic infix operators (with precedence rules), and many of the elements of the Algol control structure (with *goto* specifically excluded). These were chosen as representatives of good management tools from the realm of general purpose languages. The PDP-10 hardware model accepts these constructs with very little overhead which makes them sound technical tools as well. The remainder of Bliss is composed of operators, control structures, data structures, etc., which although not entirely unique, are somewhat different from those in other languages because: (1) of the structure of the PDP-10, (2) of the efficiency problems imposed by implementation languages in general (that is, a concerted effort was made to minimize the discrepancy between the logical Bliss machine and the physical PDP-10), and (3) no suitable models for certain management tools could be found in existing languages.

As stated in the introduction this paper is not intended to be a definitive description of Bliss. However, two aspects of Bliss related to management issues are discussed below to illustrate how these may manifest themselves in a language design:

(1) Control Structures: Other than subroutines and co-routines, the control structures of Bliss are a consequence of the decision to eliminate the *goto* (see References 4, 5, 6 for a discussion of the reasons behind this decision). In Reference 4 the author analyzes the forms of control flow which are not easily realized in a simple goto-less language and uses this analysis to motivate the facilities in Bliss. Here we shall merely list some of the results of that analysis as they manifest themselves in Bliss.

   (a) A collection of 'conventional' control structures: Many of the inconveniences of a simple goto-less language are eliminated by simply providing a fairly large collection of more-or-less 'conventional' control structures. In particular, for example, Bliss includes: conditionals (both *if-then-else* and *case* forms), several looping constructs (including *while-do*, *do-while*, and stepping forms), potentially recursive procedures, and co-routines. While anything in addition to the *goto* and a conditional branch may be considered "syntactic sugar" in most languages, these additional forms are essential to convenient programming in Bliss (although they are not all theoretically needed for completeness, see Reference 6).

   (b) Expression Language: Every construct in Bliss, including those which manifest explicit control, are expressions and have defined values. There are no 'statements' in the sense of Algol or PL/I. It may be shown[6] that one mechanism for expressing algorithms in goto-less form is through the introduction of at least one additional variable. The value of this variable serves to encode the state of the computation and direct subsequent flow. This is a common programming practice used even in languages in which the *goto* is present (e.g., the FORTRAN 'computed *goto*'). The expression character of Bliss is relevant in that the value of an expression is a convenient implicit carrier of this state information.

   (c) Escape Mechanism: Analysis of real programs strongly suggests that one of the most common 'good' uses of a *goto* is to prematurely terminate execution of a control environment—for example, to exit from the middle of a loop before the usual termination condition is satisfied. To accommodate this form of control, Bliss allows any expression (control environment) to be labeled; an expression of the form "*leave* ⟨label⟩ *with* ⟨expression⟩" may be executed within the scope of this labeled environment. When a *leave* expression is executed two things happen: (1) control immediately passes to the end of the control environment (expression) named in the *leave*, and (2) the value of the named environment is set to that of the ⟨expression⟩ following the *with.*

(2) Functional Decomposition: An effective program management technique is to insist on functional decomposition and isolation of tasks. Technical issues suggest several alternatives for constructs all of which can be considered "function like": full-blown Algol functions (with display mechanism), Bliss "routine" (without display mechanism), co-routines, macros and the (Bliss) data structure mechanism.

   (a) Functions and routines are defined and called in Bliss in a manner similar to that in Algol, except that there are no specifications and all parameters are implicitly call-by-value. Functions and routines are examples of choosing well-known and admired managerial tools and adapting them to satisfy the technical requirements of a system implementation language.

   (b) Co-routines are often used (unwittingly) by programmers in any language; their essential nature is that they preserve some sort of "status" information upon exit and continue execution upon recall based on that status. If the status becomes arbitrarily complex, the only way to retain it is to remember essentially everything which pertains to the Bliss model in the machine for a running program, i.e., the stack, declarable registers, and program counter. Such information is best dealt with by the compiler (i.e., a minor implementation change might have drastic effects if everyone using co-routines of this complexity were saving status information differently); thus the construct was included in the language.

(c) The Bliss structure mechanism allows the user to define an accessing algorithm—that is, the algorithm to be used to obtain the address of an item in the structure. In fact, there are no "built-in" data structures; the user must define the representation of every data structure by supplying an accessing algorithm for it. Once an accessing algorithm has been defined, it may be associated with a variable name and will be automatically invoked when that name is referenced. Thus, the user may choose the most appropriate (efficient) representation and may change the representation as the use of the data structure evolves.

With 20/20 hindsight it is obvious that it is the managerial issues, and not the technical ones which are the most costly, provide the most compelling reasons for adopting an implementation system, and hence are the primary ones to which such a system must respond. Moreover, a language can only make technical responses to these issues, and cannot respond to the entire spectrum of managerial issues. Conversely there are a set of issues to which the most appropriate response is at the language level.

The technical responses made in Bliss, such as the structure mechanism and removing the *goto*, are, for example, both good and made at the appropriate level. We consider the Bliss structure mechanism, for example, to be a "good" management tool because it decouples those decisions concerning the representation of a data structure from those decisions concerning the manipulation of the information contained in the structure. (In this context we consider the data structuring mechanisms of most languages to be "bad" management tools in that the representation decisions are made at a totally inappropriate time—namely, when the language is implemented.)

MANAGEMENT FUNCTIONS PROVIDED BY A 'TOTAL' SYSTEM

We wish to distinguish between two notions which the term 'system' might connote; for want of better terminology we shall refer to them as *internal* and *external*. By *internal* we mean those facilities which must be provided by a system coextant with that written by the programmer. Conversely, by *external* we mean those facilities which are never coextant with the user's program. Dynamic storage management and virtual memory systems are examples of the internal variety; editors, loaders, and linkage editors are generally of the external variety. Of course, there are numerous examples—the TSS dynamic loader, for instance—which cross this boundary.

External facilities

In some ways these appear to be the most mundane of those facilities which might be provided by a 'total' system. Editors, file systems, loaders, etc., are familiar to us all and that familiarity is indeed likely to breed a certain level of contempt—or at least a strong temptation to "make do" with the facilities that happen to be available.

However, measured against the definition of a 'good' management tool given above, most of the editors, etc., with which we are familiar are inadequate. Moreover, they are unlikely to become adequate unless invested with more specific knowledge of the structure of the items with which they deal. In particular, the notion of decoupling decisions carries the collateral notion of distributed definition and use (related definitions are grouped rather than related uses). Present editors, for example, simply do not cope with such structures—particularly if definition and use are in separate files. Fortunately, there is an excellent extant example of a system with many of these properties designed by Englebart, et al.[7]

Internal facilities

The class of facilities we have called 'internal'—those which require coextant support—are certainly more glamorous than the external ones. Our experience using Bliss strongly suggests that some of these mechanisms would be very valuable, in particular: incremental compilation, debugging at the source level (as with conversational languages), execution of incomplete programs, virtual memory, etc.

All these mechanisms represent 'good' management tools, when described at this level, in that they permit certain classes of decisions to be decoupled. The ability to execute incomplete programs, for example, is an attractive facility for permitting parallel construction of systems by several implementors.

Unfortunately there are no extant examples of systems which provide wholly 'good' tools from either the technical or managerial standpoint; the existing systems fail for two reasons:

1. They are inefficient in specific cases. To date these systems have generally been interpretive;

while a technical solution to this exists,[8] it is not clear that the residual distributed effect of this flexibility can be totally eliminated.

2. They imply binding certain decisions at a very early stage, namely, when the supportive system is written. This is by far the more serious problem. Internal facilities are efficacious to the extent to which they can presume a particular structure in the system they support. (While this is also true of external facilities, in the latter case the assumptions are purely formal.) These presumptions are inviolate and their presence clashes with our definition of a good management tool.

The position taken by Bliss with respect to internal system facilities is the extreme one, and the original rationale for it is probably fallacious: the code produced by the Bliss compiler requires *no* run time support. The rationale for this position was that some systems would be written in Bliss could not presume such support—notably the lowest levels of an operating system. While this is indeed true, the fallacy is that the majority of programs written in Bliss have not been operating systems, nor will they be.

It is possible, of course, to write one's own support in Bliss, and a fair variety of these packages have been written—one of which is worth special mention.

The "timing package"[9] is a set of Bliss routines which may be loaded with any Bliss program. Using its knowledge of the run-time structure of Bliss programs the package can intercept control at "interesting" points, notably routine entry/exit, and record various information. In particular, the usual information gathered is the frequency and duration of routine executions and the memory reference pattern.

The timing package is a "good" management tool in the sense of the definition above in that it permits postponing (a programmer's) concern over specific local efficiency until there is evidence that local efficiency has global significance. Moreover, the timing package is a good technical tool in that its presence is not presumed and there is no distributed (or local) inefficiency implied by its potential use. Most of the systems written in Bliss have been "tuned" using this facility and the results are much as one would expect: a very small portion (less than 5 percent) of a program usually accounts for most of its execution time, the programmer is usually surprised by which portion of the program is taking the most time, and improvements by a factor of two in execution speed by relatively simple modifications are common.

The example of the timing package points out both an important distinction and a language requirement not discussed previously. The distinction is between those supportative facilities whose presence and form is requisite and presumed, and those facilities which, if available, may be exploited. The language requirement is that the link to these (optional) facilities should be 'natural.' Thus, for example, we consider dynamic storage management to be an inappropriate *presumed* facility, at least in the context of a SIS, because:

1. it undoubtedly implies a distributed overhead which is intolerable in specific cases,
2. it implies binding a decision, namely a particular storage discipline, which will be inappropriate in specific cases.

Yet, the ability to define and use a dynamic storage management system, and to do so 'naturally' in the language, is totally appropriate.

SUMMARY

The results of our experiences in using Bliss for over three years for a number of large software projects reinforces our view that the major problems of software development are what we have termed 'managerial' in nature, and not technical. The use of any higher-level language can alleviate certain of these problems, a careful language design can alleviate more and there are certain features which must be provided at the language level, but there are limitations to what can be done in any statically compilable language.

Many, if not all, of the issues which cannot be addressed by a compilable language may be addressed by a comprehensive system of which a language is only one part. Some of the facilities of such a system would deal primarily with various external representations of a program. Although these facilities need to be carefully integrated, they would presumably be related to familiar facilities. Moreover, such facilities are relatively 'safe' in that they deal primarily with the formal (syntactic) aspects of a program. We regret not having paid more attention to these facilities at an earlier stage of the Bliss effort.

Another class of facilities which might be provided by such a system relate primarily to internal representations of the program and must coexist with this representation. This class is at once more glamorous, potentially more useful, and more dangerous. The utility of such facilities is directly related to their specific knowledge of the internal structure of a program. To the extent to which the presence of such facilities forces a specific set of representations, whether of data or computation, they can magnify the problems they were meant to solve.

REFERENCES

1 W WULF et al
Bliss reference manual
Computer Science Department Report Carnegie-Mellon University Pittsburgh Pennsylvania 1970

2 W WULF D RUSSELL A HABERMANN
Bliss; A language for systems programming
Communications of the ACM 14 12 December 1971

3 W WULF et al
Reflections on a systems programming language
SIGPLAN Symposium on System Implementation Languages Purdue University October 1971

4 W WULF
Programming without the Goto
Proceedings of the IFIP Congress 1971

5 E DIJKSTRA
GOTO statement considered harmful
Communications of the ACM (letter to the editor) 11 3 March 1968

6 W WULF
A case against the Goto
Proceedings of the ACM National Conference 1972

7 D ENGLEBART W ENGLISH
A research center for augmenting human intellect
FJCC 1968

8 J MITCHELL
The design and construction of flexible and efficient interactive programming systems
PhD Thesis Carnegie-Mellon University 1970

9 J NEWCOMER
Private communication

The CPM-X—A systems approach to performance measurement

by RICHARD J. RUUD

Allied Computer Technology, Inc.
Santa Monica, California

INTRODUCTION

Hardware monitors of one kind or another have been employed in the instrumentation of general purpose data processing equipment since the early 1960's. Until 1969, however, the production of hardware monitors was, in the main, confined to computer manufacturers and research projects. In 1969, the performance measurement industry was created through the introduction of relatively standard commercially available hardware measurement products.

With the advance of the state of the art of measurement and the increase in sophistication and skill of the user from 1969 to the present, it became necessary to design and build a new generation hardware monitor. This monitor is known as the CPM-X.

This paper is divided into two parts. The first section will discuss the goals and rationale involved in specifying the CPM-X. The second part will discuss the architecture and implementation of the CPM-X in light of the goals previously discussed.

SYSTEM DESIGN GOALS

The basic monitor

The CPM-X's minimum design goal was to provide the user all of the standard features normally expected on a commercially available performance monitor. Such a system is illustrated in Figure 1.

The basic features provided with this typical system consist of a number of probes and probe receivers to transmit the data from the host computer to the monitor. The data are then routed through a plugboard containing a complement of combinatorial logic and then routed to a set of counters for event counting and timing. Periodically, the data contained in the counters, together with a real time clock and other optional data for message stamping, would be written to an output media which is almost universally magnetic tape. Display capability is provided through a luminescent binary or digital readout to allow the operator limited real time monitoring of measurement experiments. The display is also useful for checking out plugboard wiring and in diagnosing monitor failures. The last essential part of this typical system is a data reduction program which takes the magnetic tape output and condenses and formats it, customarily on the data processing system which is being measured, for the user's analysis. These functions constitute the minimum acceptable capability in hardware monitor implementation.

Parallel input

A majority of monitors in use today have an additional input source to the serial inputs first described above. This is a parallel input source accepting from 18 to 36 bits of data, customarily from a host computer register such as the instruction counter. In some monitors, this data is presented to a comparator for comparison against preset, user-entered values. The results of the comparisons are then presented to standard time/event counters.

The other common use of the parallel input feature is as input to a distribution unit. The distribution unit consists of an arithmetic and logical unit and a set of storage registers. These registers are normally maintained in a core storage unit with an average size of 1,024 16-bit words. The distribution unit is capable of operating in two modes. In distribute mode, the high order 9 bits of the parallel input stream are used as an index value to address the storage unit. When a word is selected by this index value, it is incremented by 1, thus providing a count of the number of incidences of

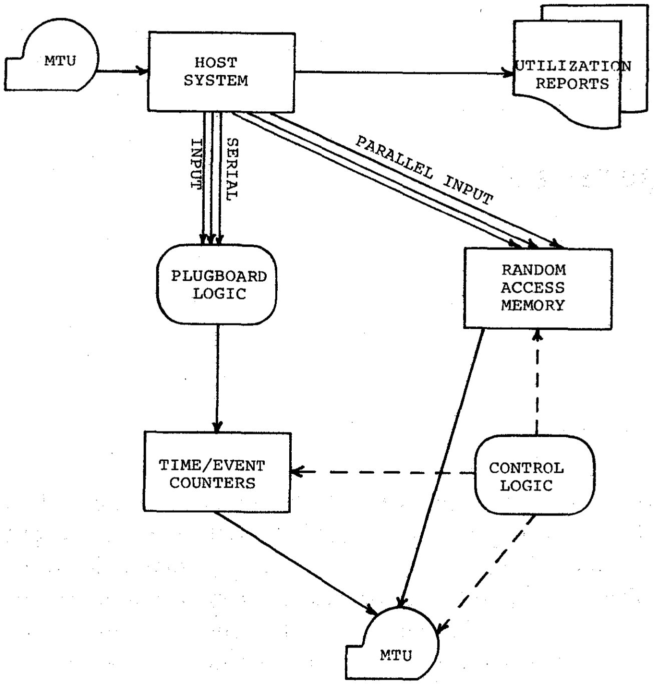


Figure 1—Representative commercial hardware monitor

each 9-bit number recorded by the monitor. In store mode, on the other hand, 16 bits of the parallel input are treated as data and stored directly into ascending word locations in the storage unit. In both modes, contents of the storage unit are periodically written to magnetic tape for off-line data reduction and report preparation. In the implementation described, 512 words are receiving data while the other 512 words are written to tape.

Additional functions

It was felt that in addition to the minimum capabilities first specified, and the parallel input capability, certain other features were desirable and necessary in the design of the CPM-X. These features had been implemented on certain experimental monitor systems, but had not been provided to the commercial monitor user.

First, the user must have the opportunity to reduce the monitor data stream to delineate points of interest and display them in real time from the CPM-X. This is not to say that there is no longer a necessity for logging of detailed data on magnetic tape for off-line report preparation. However, the user must be given an opportunity to react to his measurements in real time and to interact with the measurement system, thereby modifying his experiments as the environment changes.

Second, the user must have the ability to control, either directly or indirectly, through the results of the measurement, the logic of the monitor system. The semi-hard-wired approach of using only a plugboard to control the logic is no longer an acceptable answer.

Third, the monitor must have the ability to capture from the host computer data concerning programs in operation on the host computer in real time. The problem of matching software data to hardware data is a difficult one; yet, it must be accomplished. It is not enough to know merely how a system is performing, one must also know why it is performing the way it is. It is extremely difficult to make these judgments based solely on hardware data; just as it is extremely difficult to make them solely on software data. The CPM-X, therefore, was to be given the ability to interact with the host computer software.

Finally, the interrelated design goals of modularity and flexibility were extremely important to the design. If the wide variety of features specified above were to be made available, it would be necessary to design the CPM-X both for ease of configuration and potential growth without necessitating either major modifications to the system or that it be traded in. The flexibility requirement is mandated by the same set of facts as modularity, not to mention the need for attaching the CPM-X to a wide variety of host computer systems. It has been our experience, as suppliers of measurement equipment, that users discover new applications for hardware monitors every day. Therefore, to make it an effective tool, it was necessary to avoid being locked in, but rather to provide open-ended hardware and software solutions to measurement.

SYSTEM ARCHITECTURE

Overview

The basic system data flow of the CPM-X is depicted in Figure 2. The building blocks of the CPM-X are four basic modules: instrumentation, control, memory, and computation. The instrumentation module contains the probe receivers, plugboard, counters and parallel input interfaces. The core memory may range from 8K to 65K 8-bit bytes depending upon the configuration and needs of the user. The memory has three ports and therefore may be accessed by the control module, the computation module, or the direct memory access channel (DMA). The control module contains registers and an arithmetic and logical unit which are used to control the data flow between the instrumentation module and core memory and also to control the magnetic tape drives in the system. Similarly, the computation module has an arithmetic and logical unit and a set of registers which can be used for operator interaction in the on-line data reduction and display function. This module is also used to control all I/O devices other than the tapes, such as display terminals, teletypes, and remote terminals. It can also read and write certain registers contained in the instrumentation module to modify the set-up of the measurement. Although the DMA channel communicates directly with memory, the channel is under the control of the computation module. Interrupt lines are maintained between the computation module and the control module to synchronize their operation.

CPM-X is configured in three basic models. Model A consists of an instrumentation module, a control module, 4K of memory, and one magnetic tape unit. This configuration will provide the basic functions enumerated in the earlier discussion of the typical commercial hardware monitor. This excludes the parallel input function. The Model B consists of the same modules but the memory is expanded to provide for parallel input as described above. In addition, certain field modifications are made to the control module to enable it to perform the distributive control function. A Model C, which is the largest configuration available, adds the computation module to the above. The number of tape drives may then be increased from one to four, if desired. At least one operator's console is required which may be a Model 33 Teletype or a CRT display with a teletype compatible interface. Provision is also made for remote hard copy and CRT terminals in addition to sophisticated local CRT terminals with graphic capability.

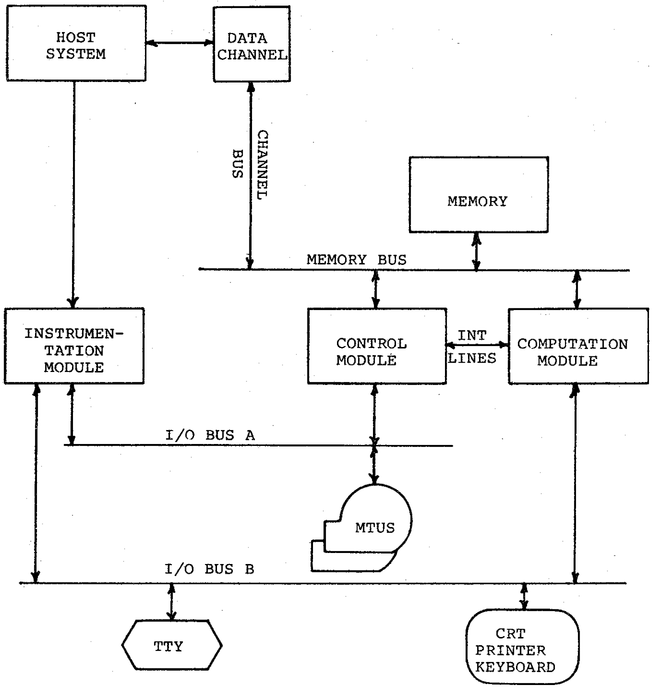


Figure 2—System data flow—CPM-X

The micro-processor

It became readily apparent that the control and computational functions of the CPM-X could most readily be implemented, and at the lowest cost, through the use of minicomputers, since both required arithmetic and logical capability and high speed register storage. It was also necessary to find a mini-computer which employed a three-ported memory because of the design considerations of the CPM-X.

The mini-computer selected was the Micro 1600-D dual processor system manufactured by Microdata Corporation which incorporated all of the required features. The 1600-D system consists of dual processors sharing a core memory which can vary in size from 4K to 65K 8-bit bytes. The system also had provisions for attaching a DMA channel to its memory bus. Each processor had a separate party-line, byte I/O bus communicating directly with its data flow.

An extremely important, perhaps over-riding, consideration in the selection of the Micro 1600-D was the ability for the implementers of the monitor to easily micro-program the processors, rather than depending on the manufacturer of the mini to perform that task. Between 250 and 4,096 16-bit words of ROM may be attached to each processor. Although cost considerations dictated that permanently written read only memory be utilized in field versions of the CPM-X, a writeable control store, exhibiting the same timing characteristics as the read-only store used in the field version, was available on the engineering model and used extensively for debugging the firmware; i.e., microprogrammed algorithms. This writeable control store together with a micro-assembler and micro-simulator greatly aided in the development of the CPM-X.

The basic firmware development strategy was as follows. Processor A, as identified in Figure 3, served the function of the control module. Various algorithms such as counter update, counter concatenate, and distribute, were developed in firmware. Additionally, a comprehensive set of microdiagnostics was developed to test the control module, the instrumentation module, and the tape units.

Processor B, the computation module, on the other hand, was primarily implemented to use the manufacturer's standard, general purpose instructions. How-

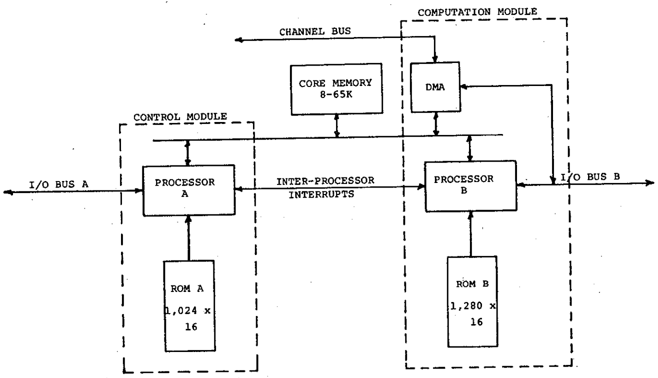


Figure 3—Application of the Microdata 1600-D multi-processor to the CPM-X

ever, this instruction set was modified to provide additional instructions, such as 32-bit multiply divide, an interrupt scheme more responsive to the environment in which the CPM-X would be used, and the various firmware routines for controlling the DMA channel interface with various manufacturers' host computers.

Counter hardware

A CPM-X may contain from 16 to 64 counters. Most counters have but a single function which is to accumulate time or event data from serial probe inputs. Certain counters, however, can perform other functions in addition to that of time/event measurement. The additional functions that these multi-function counters can perform are: Parallel data input to the CPM-X, distributions, and comparisons. The packing of the CPM-X counter groups is such that a standard group contains eight single-function counters. A multi-function counter group, on the other hand, contains four counters, one of which is a standard, single-function counter; one may be used as a 24-bit parallel input and shift register; while two may function as 24-bit comparators.

Figure 4 illustrated the data flow of a multi-function counter group. Since Counter A, in the diagram, functions in exactly the same manner as any other single-function counter, a description of the multi-function counter group, with references to the counters shown in Figure 4, will suffice to describe all possible counter combinations in the CPM-X.

All counters can receive data from the time or count serial input hubs on the plugboard. The count hub merely causes the input signal to be sent directly to the low order bit input of the counter; thereby accumulating a value in the counter equivalent to the number of times the signal has changed state from a logical 0 to a logical 1. The time hub takes the same input signal and uses it to gate the output of an internal or external clock. Therefore, these clock pulses will accumulate in the counter so long as the input signal is in the logical 1 state; effectively turning the counter into a timer.

An additional set of inputs is available for Counters B, C, and D, the multi-function counters shown in Figure 4. This is a 24-bit parallel input bus driven by 24 probes. Counters B and C may also obtain input parallel by bit, serial by byte from the 8-bit I/O output bus of the control processor.

The buffer outputs are available to the control processor parallel by bit, serial by byte on its I/O input bus. In addition, the results of comparisons, when multi-function counters are used in the compare mode, are available at their respective high-low-equal hubs on the plugboard.

In addition to the logic shown in Figure 4, there is control logic which is not illustrated. The control output logic is conditioned by the decoding of the control processor's I/O control register. This logic differentiates between address and data cycles and sets read or write status for subsequent data cycles. When a counter needs service, either because there has been an overflow from the high order bit position or a strobe pulse has been encountered in the case of the parallel input counter, the address of that counter is generated by hardware logic; and an interrupt line is raised to the control processor. This enables the control processor to either perform a counter update

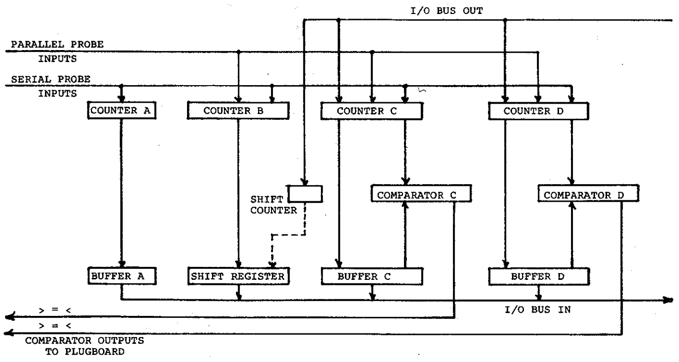


Figure 4—Multi-function counter group

sequence, a distribute sequence, or transfer the information contained in the parallel counter to storage.

Standard counter operation

Standard counters, or multi-function counters operating in the standard mode, accumulate serial input data at a maximum rate of 20 million counts per second. Operating at this maximum rate, the counter will overflow in approximately 3.3 milliseconds. When an overflow occurs, the address generation logic presents an interrupt to the control processor together with the address of the overflowed counter. The processor firmware then utilizes this information to update a 16-bit counter extension contained in core memory; thus, a 32-bit counter is fashioned with the low order 16 bits being implemented in external hardware, and the high order 16 bits being implemented in core storage. The buffer follows the counter and always reflects the exact contents of the counter, except during a dump operation.

When a snapshot of counter contents is to be taken for manipulation, display or recording purposes, the dump sequence is entered. First, the connection from all counters to their associated buffers is broken at the same time by the control processor. This has the effect of stopping the input to all buffers while allowing the counters to continue to accumulate data. The buffers can now be unloaded without effecting data integrity and any problem of time skew is eliminated, since the buffers were all stopped at the same time. The control processor then unloads the two bytes of each buffer, concatenates these two bytes with the high order two bytes of the counter contained in core and stores these four bytes in another area of memory reserved for the purpose. When this update process is completed, the connection between the counters and the buffers is re-established allowing the buffer to be updated to the current counter value. The control processor then presents a microprogrammed interrupt to the computation processor in a Model C. This results in software interrupt being initiated in that processor. In the case of a Model A or B, the completion of a dump sequence initiates a firmware tape-write sequence, thereby recording the accumulated counter data on magnetic tape.

Comparator operations

Counters C and D shown in Figure 4 can also be used as comparators. In this mode, the width of the counter is expanded from 16 to 24 bits. The counter is first loaded by the control processor with a 24-bit comparand. The write sequence automatically causes this information to be transferred from the counter to the buffer where it is held. When the comparator receives a compare strobe signal from the plugboard, the information contained in the parallel probe inputs is gated to the counter; and the contents of the counter are then compared to the contents of the buffer. The high-low-equal result is then latched at the output of the comparator and sent to the plugboard. The result of the last compare is available until a new compare cycle is initiated. Comparators are extremely useful as filters in distribution and save and store mode, as well as being used solely as comparators.

Parallel input operations

Counter B, also 24 bits in width, may be used as a parallel input register to the CPM-X. As in the case of the comparators, 24 bits of data will be gated into the parallel input register upon receipt of a strobe signal from the plug-board. Two events then take place. The interrupt line is raised to the control processor and the address of the parallel input register is presented when the interrupt is acknowledged, allowing the processor to locate the register for unloading. At the same time the processor is being interrupted, the information contained in the counter portion is transferred to the buffer and concurrently shifted right under control of the shift counter. The shift counter is loaded from the processor's I/O output bus, and the count remains constant unless specifically changed. The truncated results are then read from the shift register portion of the counter into the control processor's data flow. The shift function is used to provide an appropriate window in distribution and data storage functions.

The channel interface

In order to close the loop between the host system software and the hardware monitor, the CPM-X, Model C, employs a data channel interface. The interface is unique to the host computer system to which it is attached and therefore requires changes to read-only memory and the hardware when moving from computer to computer.

The channel interface, quite simply, makes the CPM-X look like a standard peripheral device to the host system being monitored. As an example, the attachment of a CPM-X to an IBM System/360 or System/370 may be considered. The CPM-X is assigned a control unit and device address by the installation and is attached to a system data channel through the standard I/O interface connector cables. The monitor's priority is dependent upon the physical position in which it is attached to the channel relative to other control units on that channel, and may be varied to suit the installation's requirements.

Data transfer operations are commenced when the host system issues a start I/O instruction. The CPM-X is selected and the command code is decoded by firmware routines contained in the read-only memory of the computation module. The DMA channel controls, indicating read or write status, beginning memory address, and length, are then set up. Once this is accomplished, the host processor is connected to the DMA channel for the data transfer operation. Since the data is transferred directly to the CPM-X's memory, the rate of transfer can be adjusted and data overruns will not occur.

At the end of the data transfer phase of the operation, signalled either by the expiration of the count in the host computer or an end-of-buffer condition in the CPM-X, the appropriate status indications are generated by the firmware and presented to the host processor so that the I/O operation may be terminated.

In order to prevent the host processor from sending more data than the CPM-X can handle because of software or performance limitations within the computation module, the channel-end and device-end bits are not presented at the same time. When an operation is completed, the channel-end bit is routinely returned with ending status by the firmware. However, the software program must issue a specific instruction to present device-end to the host computer. This has the effect of making the CPM-X appear busy to the host computer should it wish to initiate additional data transfers before the CPM-X software is ready to accept them.

Should conditions occur asynchronously in the monitor which require that the host processor be alerted, CPM-X software can issue an attention interrupt. Upon recognition of this interrupt, the CPU can initiate a read operation to the CPM-X to determine its cause.

The channel interface provides a complete two-way communication path between the host processor and the CPM-X. Although the IBM 360/370 interface has been described, the principles enumerated are applicable to other manufacturers' hardware and channel interfaces are currently offered for both Univac and RCA computers.

Plugboard control

In order to provide communication between the hardware of the instrumentation module and the software of the computation module, the plugboard control interface was developed for the CPM-X. This gives the computation module the ability to modify hardware setups and also to recognize the occurrence of significant hardware events. There are two sources for output data and two sources for input data.

The command register is used as a source of pulse information from the computation module. Its 8 bits are set by command byte and remain set for a period of 200 nanoseconds. The outputs are used where momentary signals are required for such purposes as setting or resetting latches and resetting counters to zero.

The program register, on the other hand, while also 8 bits wide remains set to the bit configuration with which it was last loaded until reloaded by the computation module. Primary use of the program register is to modify the hardware setup by conditioning and deconditioning AND gates which in turn will alter the manner in which data or control signals are routed through the plugboard.

One source of input to the computation module is the multi-function register. This register is 16 bits wide and is read into the computational processor on demand. Inputs to the register may be probes for statistical sampling operations or any other signal available at the plugboard.

Finally, a variable number of interrupt hubs are provided which, when impulsed, cause the computational software to interrupt to a fixed location. These interrupts are used by the software to recognize exceptional hardware events occurring in the host processor for recording or monitor action.

Software support

The CPM-X, Models A and B, do not require internal software support. All algorithms required for operation of the control processor are contained in its ROM and core is only used for data storage. The Model C, however, will have a complete monitor operating system. All models of the CPM-X are supplied with versions of the Measurement Summary Report program to enable the user to perform data reduction and report writing functions on the host CPU.

The basic software design is that of an interactive interpreter. Through it, and some of the hardware features described above, the user will be able to exercise control over all aspects of the measurement system. The software interface will either be through a teletypewriter or teletypewriter compatible CRT terminal.

The system is written in assembly language and occupies 3,000 bytes of storage. It is expandable and routines can be added by the user as required; however, full user capability for on-line data reduction and display is provided by this basic system.

The software support provided for the host computer to utilize the channel interface is limited. In the case of IBM OS support, the channel interface must be programmed at the EXCP level. Error routines for the CPM-X will be provided for inclusion in the operating system. The level of implementation for other manufacturers' host processors will be similar to that provided for IBM.

Parallel input applications

This family of applications will be described in some detail since they best illustrate the interaction of all elements of the CPM-X. These applications can be performed with a Model B or a Model C; however, limits and buffer size must be set manually in the Model B configuration. The applications described will therefore cover the Model C to illustrate the use of the computation module.

The basic strobe signal used to gate data into either the comparators or the parallel input register is usually obtained from a host computer signal which indicates that the register being inspected by the monitor has just been changed and that the new data is valid. Generally, comparators use this raw strobe signal so that every time a register is changing, a new comparison is being made; and the output latches are set accordingly. These latches may be used as direct inputs to counters to indicate the frequency with which certain data are occurring or their time duration. The outputs of multiple comparators can be wired together through plugboard logic to provide high and low limits and to indicate when data fall between these limits. This wiring scheme is usually used when comparators condition the strobe input to the parallel data register. Thus, limits are set by the computation module's software and the only time that data is accepted by the parallel input register is when that data falls within the range defined by those comparators.

Store mode

An area of main storage is allocated by the software as a data buffer. This storage area is divided in half to provide for double buffering. The word size of the buffers may be either one, two, or three bytes, depending upon the needs of the experiment. In a store mode operation, where a three byte word size is indicated, the total contents of the 24-bit parallel data register are transferred to the shift register and then to main storage. Each register transfer initiates a storage cycle whereby the data element is stored sequentially in the main storage buffer until that buffer is filled. Then an automatic buffer switch takes place and the software is alerted that a buffer has been filled. Store mode allows any sequence of data elements to be recorded for future analysis by the CPM-X. These elements may consist of instruction addresses for detailed trace operations, instructions themselves, data elements, or any other data stream the user might wish to specify.

In order to economize in both storage and speed, the user may wish to specify a basic data element of less than 24 bits. In this case, either the low order two bytes or the low order one byte of data is transferred to main storage. To enable any 8 or 16 bits out of the 24 to be recorded, a value is loaded into the shift register causing the specified number of bits on the right-hand side of the data element to be truncated; thus, any contiguous portion of the parallel input register can be recorded.

It is often desirable to record data only when it falls within defined limits. For this type of filtering, the comparators are employed. Two comparators are set with the upper and lower bounds, and the comparators' outputs are so wired as to only allow the parallel input register to be strobed when data falls between them.

A more complex variation of the store mode is that of sequencing, in which the experimenter is interested in retaining a previous event; but he only wishes to record it if one or more predefined, subsequent events take place. The common usage of this mode of operation would be in determining the location from which a sub-routine is being called. When a sub-routine of interest is loaded into core, the address of its entry point can be transmitted over the channel interface to the CPM-X; it is then loaded into a comparator. The plugboard is wired so that every address is recorded by both the parallel input register and the comparator. The address contained in the parallel input register is transmitted to the shift register and held there. When the next address in the stream arrives, it is compared to the contents of the comparator. If a match exists, the parallel input logic is allowed to interrupt the control processor and, thereby, transfer the contents of the shift register to main storage. This has the effect of transferring the calling address to main storage, if that particular subroutine is entered. Multiple sequencing can be effected by using multiple comparators and cascading their output through latches.

Distribution

The second mode of use for the parallel input register is that of distribution. In this mode, a portion of the data ingated to the parallel register is used as a displacement in addressing main core. In this case, the buffer area in main core is divided into a number of logical accumulators. The accumulators may be 8, 16 or 32 bits in width. A buffer area is then defined by the user which must consist of a number of logical accumulators which is an integral power of two. Two such buffers are required to provide for on-line data reduction or transfer to tape without loss of data.

Distribution data are normally storage addresses although other data, such as operation codes may be used. Considering the case of storage addresses, it is often of greater interest to evaluate in detail the distribution of references to one or more sub-divisions of main storage than to make a gross evaluation of all of the storage available on the system. When it is desirable to sub-divide storage in this manner, the comparators are employed, as in the case of store mode, to define the boundaries of the storage area to be inspected. In the following discussion, L1 will indicate one boundary and L2 the other boundary of such a sub-division.

In many cases, it is impossible to allocate logical accumulators to individual data elements on a one-to-one basis because of the mis-match between the size of the memory area to be inspected and the amount of CPM-X core memory available for accumulators. When the number of discrete storage addresses exceeds the number of logical accumulators available, the occurrence of two, or more, adjoining data addresses must be summed in a single accumulator. A parameter known as the Resolution Factor has been defined as the number of data element occurrences which are summed in a single logical accumulator. A Resolution Factor of unity is ideal and the resolution of an experiment is inversely proportional to the Resolution Factor.

The boundaries of the storage area to be inspected are often dictated by factors beyond the experimenter's control, such as the size of a given user program or the executive. The experimenter is then generally interested in examining this area with the highest possible degree of resolution. The shift counter of the Parallel Input Register is employed to accomplish this. The value of C to be placed in the shift counter is calculated by the computation module based on the parameters of the experiment and is defined as the whole number less than or equal to:

$$C = 1 + \text{LOG}_2 \left(\frac{|\text{L1} - \text{L2}|}{\text{NUMBER OF ACCUMULATORS}} \right)$$

Once the shift count is computed, the Resolution Factor can be defined as: RESOLUTION FACTOR $= 2^c$.

In a typical application of distribution, the CPM-X might be used as follows: The experimenter would first enter the name of the job to be analyzed at the CPM-X console; this information would then be transferred to the executive of the host CPU via the channel interface. When the selected job is loaded into the host computer's memory, the boundaries of its core region would be returned to the CPM-X over the channel interface, together with an imperative to start address distribution. The computation module would then compute the shift count and load it, together with the boundary conditions into the shift counter and comparators of a multi-function counter group. The computation module would then order distribution to commence. At the termination of the job, the computation module would again be alerted by the host computer via the channel interface. The distribution would be wrapped up and required housekeeping performed.

CONCLUSIONS

The CPM-X design was based on three things: a survey of what was currently available commercially; techniques developed in experimental monitor systems; and the needs of the experimenter.

The key to the architecture of the CPM-X was the total integration of a duplex mini-computer into the hardware monitor. The Microdata 1600-D was selected primarily because of the ease with which it could be micro-programmed by the implementers. This decision has, in fact, materially reduced both time and cost in the development of the CPM-X. Furthermore, the utilization of interchangeable read-only memory modules, together with interchangeable hardware has greatly alleviated the problem of model changes and of interfacing the CPM-X to a variety of host computers.

Some of the more complex measurement applications have been described to illustrate the interaction between various elements of the CPM-X system.

ACKNOWLEDGMENTS

The author would like at this time to acknowledge the substantial contributions made by several individuals

in the design of the CPM-X. Messrs. Paul E. Chanel and Mark J. McGrew, of Allied Computer Technology, Inc.; both made numerous contributions to the original specifications of the CPM-X. Mr. Louis Gallenson, formerly of Allied Computer Technology, Inc., and currently a member of the Information Science Institute of the University of Southern California; contributed much to the early engineering design of the CPM-X, especially in the multi-function counter area. The overall design was carried out by Mr. Thomas O. Ellis, formerly of the Rand Corporation, and currently a member of the Information Science Institute of the University of Southern California. Finally, Mr. Frank M. Stepczyk, of Allied Computer Technology, Inc., was responsible for the definition and creation of the software routines for both the Microdata 1600-D and various host computer channel interfaces.

One of the experimental monitor systems analyzed on the CPM-X development is ADAM designed by Mr. James Hughes and his staff at Xerox Data Systems, Marina del Rey, California. Since no documentation has been published outside of XDS, the author can only thank Mr. Hughes for his discussions of ADAM without quoting a reference source for the reader.

REFERENCES

1 ALLIED COMPUTER TECHNOLOGY INC
Computer performance monitor II: systems summary manual
1969

2 R A ASCHENBRENNER et al
The neurotron monitor system
AFIPS Fall Joint Computer Conference 1971

3 G ESTRIN et al
SNUPER computer
AFIPS Spring Joint Computer Conference 1967

4 INTERNATIONAL BUSINESS MACHINES CORPORATION
IBM System/360 and System/370 I/O interface channel to control unit
Original Equipment Manufacturer's Information Form Number 6A22-6974-0 1971

5 MICRODATA CORPORATION
Micro 1600/21 Micro 821
Computer Reference Manual Form Number 71-1-821-001 1971

6 MICRODATA CORPORATION
Micro 1600
Computer Reference Manual Form Number 71-1-1600-001 1971

7 MICRODATA CORPORATION
Micro 1600-D dual processor system
Internal Document Number PS20002400 1972

System performance and evaluation—Past, present, and future

by C. DUDLEY WARNER

Computer Synektics, Incorporated
Santa Clara, California

INTRODUCTION

Data processing—Profit or loss

With the billions of dollars in installed computer equipment deployed worldwide and with vast sums needed to operate, program, and maintain this hardware, computer users are increasingly aware of the need to improve the efficiency of data processing operations.

Corporations use computers to handle many tasks. The range of tasks and size of the computers increase constantly. But corporations do not know how to evaluate the efficiency of their data processing operations. Inability to make an accurate assessment has kept management fearful of the entire data processing experience and has resulted, generally, in a hands-off attitude. This tail-wags-the-dog situation places a particularly heavy burden on that portion of management directly responsible for the data processing operation. They have to make recommendations on new and/or added equipment, but they lack objective techniques for evaluating the DP operation and projecting needs.

Such a situation would be unacceptable in the management of human or mechanical resources. People are evaluated—machines are evaluated—where appropriate, their interactive performances are evaluated. But the electronic data processing arena remains a mystery, unresponsive to the profit and efficiency standards integral to any business. Strangely enough, a data processing manager may find himself in the unique position of gaining more prestige because he requests increases in computer sizes (and costs, therefore) than because he effects sizable reductions in costs through more efficient management of current equipment.

Time-and-motion studies for computing

Data processing must be brought into perspective; it can and must be made a 'profit center' like other company operations. The DP function is similar to manufacturing in that raw material (data) goes in, and a finished product (reports) is created. In manufacturing, we find lathes, drill presses, mills. In data processing, we use tape and disk drives, readers, printers, punches, and, of course, the CPU.

What manufacturing operation has not been scrutinized in time-and-motion studies? These evaluations determine the productivity of each piece of equipment, deliver recommendations on present and planned resources, specify how particular operations should be performed to achieve maximum throughput at minimum cost (i.e., productivity).

A hardware monitor is essentially a specialized stopwatch to perform time-and-motion studies on computers. It supplies a scientific approach to data processing control. A modest investment in monitoring that can increase the effectiveness of existing computer hardware and software seems preferable to buying more hardware and hiring more people.

Versatility or efficiency?

In trying to make computer systems versatile, manufacturers and users have forgotten the need for efficiency. Most of today's systems are poorly coordinated and wasteful, with much idle time and little overlap between various system resources.

This kind of inefficiency is costly, especially in job turnaround time and system throughput. Recent statistics show that the average CPU is active less than

30 percent of the time, with many clocked at 10 percent and lower. Stated another way, some CPU's are idle 90 percent of the time! No manufacturing operation would be permitted to continue at these levels of inefficiency. Percent-of-capacity averages in U.S. manufacturing are nearer 80 percent to 85 percent in well-run operations!

THE NEED FOR MEASUREMENT

Wanted: Controls to match our systems

Today, with technical development reaching a point of diminishing returns, coordinating our system operations has become crucial. Our hardware is sleek and fast, but our efficient use of that hardware is a failure. Improving current systems control is unquestionably more important to profit than fourth generation hardware, particularly in larger installations.

Organizing a job stream to use available computing resources better is a logistics problem. Hardware resources include the CPU, memory, tapes, disks, card readers, other peripherals; software resources include assemblers, compilers, operating systems, sorts, RPG's, subroutine libraries. All these resources functioning together comprise a system. Logistical reorganization might mean improving the balance between resident and non-resident routines, obtaining more interaction between peripherals, increasing the overlap of CPU and I/O, or redistributing peak loads and other bottlenecks that degrade overall system performance.

The start: Basic operating facts

An evaluation based on facts, not speculation, is vital before system elements can be rearranged effectively. Before monitoring, intuitive methods had to be used. Even sophisticated simulations were based on theoretical models or assumed hypothetical situations.

The two basic approaches to system measurement are hardware and software. A hardware monitor is preferable because it does not interfere with or create overhead in the system it is monitoring. A software monitor is a program, and an artificial situation is created when the monitor is part of the job stream, because the monitored routines must often wait for the monitor.

THE PAST: A BRIEF HISTORY OF MONITORING

The hardware measurement devices that led to the evolution of today's sophisticated monitoring concepts are summarized in Table I.

TABLE I—Hardware Monitor Development

Year	Equipment	Source
1961	Channel Analyzer I/O channel & CPU wait time	IBM
1962	Program Monitor Full program trace on tape	IBM
1964	Program Event Counter (PEC) Channel & CPU overlap	IBM
1967	System Analysis Measuring Instrument (SAMI)—Measures all system resources, 4 tons (!), 64 counters, 32 comparators	IBM
1968	Basic Counting Unit (BCU) Transportable, limited system resource measurement, punched card output	IBM
1969	System Utilization Monitor (SUM) Portable, tape output, optional comparators, measures all system resources	Computer Synectics, Inc.

Early developments by IBM have been completely overshadowed by the rapidly evolving hardware and software monitor efforts of several smaller companies in the U.S. since the first non-IBM hardware monitor was marketed in 1969. Hardware monitors are already on a second generation iteration. IBM hardware monitor use is available only to IBM customers upon request.

THE PRESENT: MONITORING TECHNIQUES

How hardware monitoring works

Monitoring devices are sensors attached to particular signal lines to measure the presence or absence of electrical impulses. For example, a sensor could monitor a signal that is present only when the CPU is in wait state; signal absence would indicate CPU active state. Monitoring does not affect the signal or the system but, like an oscilloscope, "looks" without interference or degradation. Signals are routed into a Boolean plugboard where wired logic can produce data on combined functions such as total I/O time or I/O and CPU overlap. To illustrate, in a three channel system (i.e., two selector channels and a multiplexer channel), signals for each channel would normally be routed through an OR function and into a counter to provide a channel-busy indication.

When this composite signal is routed to an AND block and mixed with CPU-active signals, CPU and

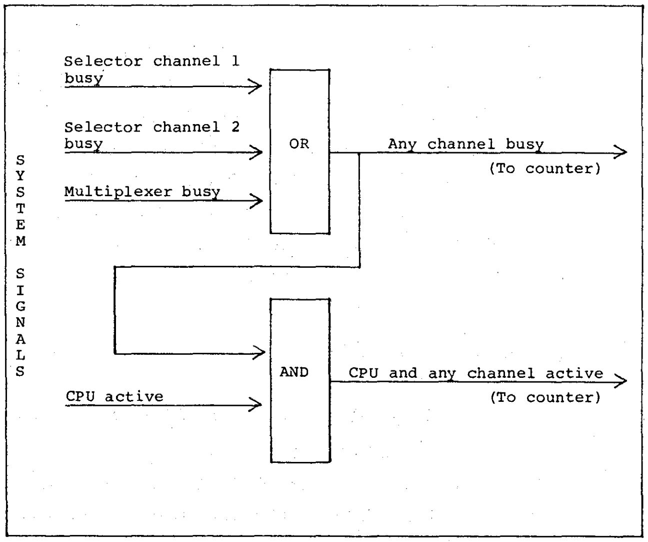


Figure 1—Routing signals

I/O overlap is indicated in another counter. Figure 1 shows a sample of signal routing.

Sensed information is summarized and recorded, and the data is then processed by a program to print measurement results. From this output, problem areas can be identified and remedial action initiated.

Useful measurement information

What kinds of data are measured: An example would be the amount of CPU active versus wait time. A CPU that waits 75 percent of the time and processes 25 percent might appear to be too large; one that processes 75 percent of the time and waits 25 percent might be too small. More operational details would be needed for meaningful conclusions.

Other relevant information would include time spent on problem programs or in the operation system. In a multiprogramming environment, regions or partitions can be monitored for task switching frequency. Instructions executed in a given time span may be measured; individual programs and routines can be timed. I/O utilization by channel, controller, device, or device component reveals CPU and I/O overlap. When there are multipaths into a system, is there high utilization in one path, none in another?

A system profile

Figure 2 illustrates a typical system profile covering times for CPU active and wait, channel utilization/wait, total and individual channel use, channel overlap, compute-only, and wait only. Just one profile item, compute-only, would be clearly affected by using a larger processor; compute-only time would decrease. Different factors determine impact on other items: channel/CPU overlap depends on job mix; wait-only time can reflect operator activity such as mounting tapes, replacing disk packs, etc.

To return to the 75/25 percent contrast, the CPU that waits 75 percent of the time may well be too large for the work load. On the other hand, it may be waiting because of poor resource utilization and overlap; the work load could, in fact, be so large and so poorly handled that management is contemplating an even larger CPU.

Real-world problem solving—Two examples

Monitoring advantages are most obvious when evaluating the potential of a larger processor for in-

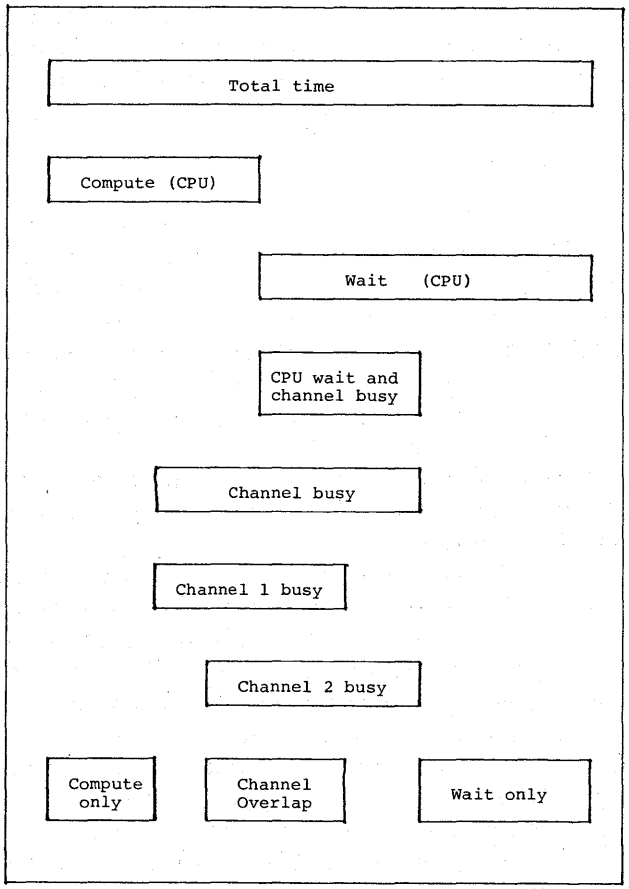


Figure 2—Measured system profile

creasing throughput. However, monitoring and analysis can also indicate whether a problem can be solved by fine tuning, and if so, how. Sometimes, identifying the problem itself is the real difficulty.

The airline

An early instance of measurement to uncover a problem occurred when an airline was trying to put a nationwide reservation system on-line. Using excellent simulation techniques to detect overload, system developers brought all regions on-line up to the busiest—New York City. It was estimated that New York accounted for 40 percent of the work load and the other regions, 45 percent. Thus, adding New York would bring the system to 85 percent of capacity. But when the switch was thrown, the system was immediately overloaded!

A monitor was installed to get the facts. It revealed that before New York was brought in, the load was really 90 percent, not 45 percent. The reservations operators didn't trust the computer. After entering reservation data, they inquired to see if the system really had that data—a double I/O load. The solution was anticlimactic—to signal the operator that the data was in, the typewriter ball was set up to 'wiggle'.

The bank

Sometimes problems themselves are elusive. An IBM customer, a bank, replaced its 7074 computers with a 360/50, got no higher throughput, and went to a 360/65—still without throughput increase. The customer, by watching the CPU wait light, determined that his seven most important jobs consumed all available CPU time.

Monitoring was undertaken for several days to profile the whole system. The seven 'capacity' jobs were indeed CPU-bound, but took only 3 percent of CPU active time. Since the CPU was active 15 percent of the total system time, these seven jobs took less than ½ percent of total system time. The problem was system operational logistics. Many short jobs required new I/O setup; the physical arrangement was crowded and disorganized. And the bank still used the 80-character record, so disks and tapes contained these short, unblocked records with gaps that induced extreme rotational and seek time delays. The bank didn't need a 360/65, or even a 360/50. When the physical problems were solved, the work could have gone on a Model 40, with plenty of system capacity to spare.

THE FUTURE: COMPUTER SYSTEMS—OVERHEAD OF PROFIT CENTER

Past efforts to monitor, measure, and manipulate the system resources and system burden (the work load) have lacked the real-time control that managers believe is vital to today's complex systems. To meet this need, monitor development is evolving toward a monitor that will act as a real time controller of resources and can thereby enable a DP operation to convert itself from creeping overhead to profit center.

Manufacturers tentative steps

Current IBM 370/165 and 370/185 models incorporate a meter/recorder that can be plugged into the Wait light to provide a continuous indication of wait time. Here is the first direct manufacturer recognition that users should expect maximum efficiency and productivity from their machines as a standard feature. But such a feature is totally inadequate for measurement needs. It furnishes a small amount of historical information—it provides no potential for real time remediation of system inefficiencies. It is not only inadequate but highly questionable. We can challenge its design and inclusion on the standpoint that the manufacturer's best interest may not be served by providing the means to self-regulate equipment or operations.

Work-scheduling possible with real-time monitors

Properly designed monitors now starting to appear in the marketplace will make possible an increasing measure of real-time monitoring and control over the system resources and work load. Such work-scheduling control is starting to take two basic forms:

1. Matching Work Load to Resources

   In this form of work-scheduling, the operations staff will be able to monitor—in real time, through a suitable console display—the actual current levels of usage of the various resources of the system. The usage levels are averaged over a range of time periods most useful to the type of work being processed. In this dynamic situation, an operations manager can call from the job stream the kinds of jobs he knows will use the resources he can see are available. This control ability presupposes his knowledge of basic job functions and their essential use of resources and his ability to manipulate the work load effectively in dynamic mode. Both these prerequisites are becoming increasingly common in modern installations.

2. Scheduling to Deadline

   Another form of work-scheduling being introduced, associated with control software as well as with real-time monitoring, incorporates the necessary or arbitrary placement of a deadline against the work load backlog. Once invoked, the system can then organize its own work load up to the deadline in the most efficient manner, or it can indicate that completion of the job stream, although optimized, is not possible. Reset to a later time, the rescheduled job stream is adjusted until an acceptable compromise is reached. This approach to work-scheduling takes the opposite view to that made famous by Parkinson, that "Work (in the computer system) expands to fill the time available for completion." The new control method says, in effect, "Can you (the system) complete this work load by time 'x' using all your resources as efficiently as possible?"

Measuring and accounting for resource use

As systems have become more and more complex in hardware and software, the problems of accounting have become increasingly complex. Accounting is a particularly difficult problem for communications-oriented systems. In systems operating solely in overhead mode, such accounting has often been arbitrary, based usually on a DP manager's need to assign costs without benefit of realistic input on cost allocation in terms of system usage. Monitors that can sample and record all essential resource utilization can, however, greatly improve the quality and accuracy of costing, accounting, and billing—a factor that is already recognized by many of the major time-sharing services and is acquiring new stature in profit-center-oriented computer users. In such applications, each individual resource must be analyzed ahead of time as to actual cost, including assignment of the internal overhead of the computer operation itself (operators, floor space, library functions, etc.). Thus established and measured, the 'computer system' as a profit center will be in a far better position to use and sell its resources against the goals of efficiency and productivity.

Hardware predicted for these goals

What form of hardware will be available to serve these new goals of system monitoring, measuring and accounting, and work-scheduling? Although it is too early to be specific, some trends are emerging. The first type of equipment, for use with smaller third-generation systems (down to 360/30 sizes), will be a compact, console-mounted monitor that will give basic resource utilization indexes as proportions, or percentages, over various time periods. Shift personnel can use this information dynamically ('eyeball' mode) to control available resources and maximize system use. Once operators become accustomed to on-line process optimization as an essential professional task, it is estimated that such a monitor should be able to increase system throughput of the average small- or medium-sized system by as much as 35 percent. These kinds of productivity gains can eliminate a complete shift in many installations.

For larger systems operating in multiprocessing or multiprogramming mode, the problem of system resource control becomes much more complex. "Eyeball" evaluation would far exceed the capabilities of the most competent and experienced operator. In such cases, a process controller is applicable. It is a hardware monitor under the continuous, real-time control of a minicomputer that will have in its own memory the complete job stream with the resources it needs. The user will have to analyze and plan the normal work load being handled by his system, so that the required system resources can be mapped and recorded in detail. Detail down to scientific core mapping is feasible. Once effected, the process controller will be able to function in essentially the same way as the process controller used in industry to handle complex manufacturing processes (e.g., petrochemical reactions involving a multitude of sensed values in temperature, pressure chemistry, flow, etc.). On-line process control of systems may seem initially to abridge the decision-making authority of computer operators. It seemed so to experienced operators of petrochemical plants in the mid-1950's. But it has already been shown experimentally that this feedback-coupled process control of a complex computer system provides huge increases in productivity and throughput.

Potential for savings

Until the computer system can be viewed as a cost or profit center, it will remain a major source of needless expenditure and low performance—which is its current status in far too many installations worldwide. To date, monitoring has often been misinterpreted by operations staffs as personally and professionally threatening. It could be of tremendous professional value in assisting a staff to develop a cost-effective computing system. Monitoring will continue to be resisted by manufacturers who have something to hide. But the history of competitiveness that has characterized the computer industry will inevitably result in the acceptance of monitoring, measurement, and control of the computer system as a first step. Use of this costly computer resource as a profit-making tool will be the immediate and desirable derivative. In the process, management will acquire again the understanding and responsibility that it abdicated ten years ago with the introduction of third-generation equipment. These are healthy advances. Uncontrolled machines consuming huge costs and personnel have no place in the modern corporation.

A philosophy to system measurement

by HENRY O. CURETON*

Hewlett-Packard Company
Cupertino, California

A PHILOSOPHY TO SYSTEM MEASUREMENT

Although a great deal of emphasis has been placed on system measurement during the past few years, each of us has his own opinion and ideas as to what it means. From a practical and immediate point of view, System Measurement is the process of obtaining useful information on the performance of software and software-controlled computer hardware. From the same viewpoint, System Measurement is the key to system efficiency. The measurement tools currently available commercially are normally artificially classed as either hardware or software, based on whether they are constructed with electronics or software elements. This particular distinction is rather meaningless to the user since what is being measured and how useful the measurements are, is of far more importance.

Efforts were directed toward the pioneering of measurement tools which would help determine the effectiveness and the efficiency of computer systems, whether they be software-controlled hardware systems or application programs. In determining whether to develop hardware or software measurement products, it was decided to develop products in the software line for the following reasons:

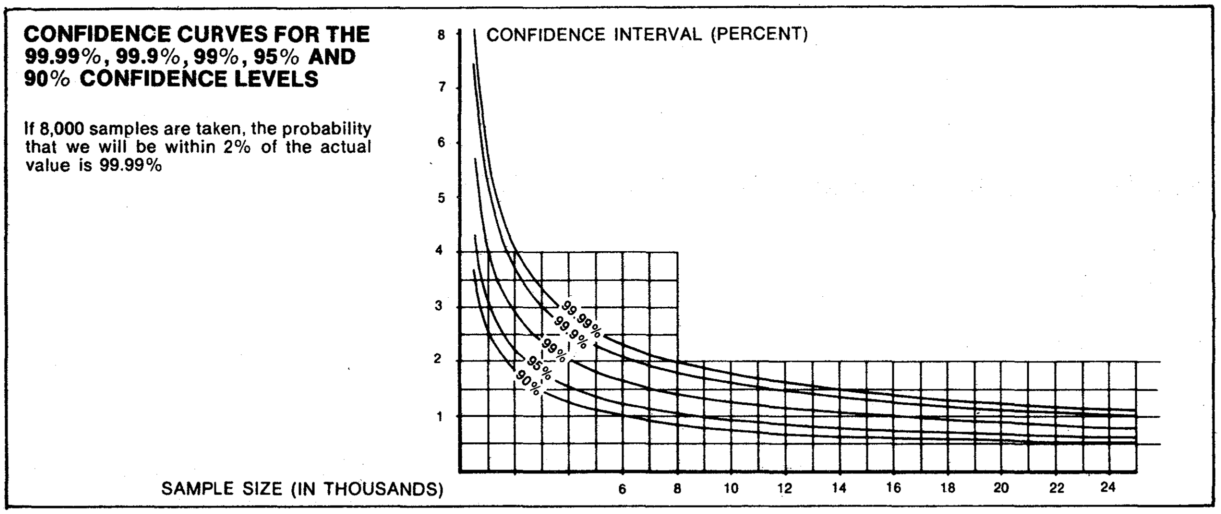
- *Greater flexibility*—Software allows many users to simultaneously measure their systems and it can be "easily" tailored to the specific needs of the user.
- *Cost*—Due to the complexity and cost of hardware components, software can be developed and distributed at a much lower cost.
- *Greater number of measures*—Hardware devices usually have a limited number of counters (32 on some commercially-available monitors) whereas software can handle as many as is practical. As an example, the Configuration Utilization Evaluator, CUE^{TM}, can concurrently monitor up to 500 different devices as well as system software characteristics such as average queue depths, number of loads of transient modules, behavior of job initiators, etc.

While recognizing the advantages of software measurement, one must also be aware of the degradation which is ever-present with software probes. Hardware monitors are external to the system and consequently, do not require any of its resources during the monitoring phase, whereas software requires CPU cycles, I/O activity and primary storage during its extraction process. Consequently, in order for software measurement to be successful, an extraction technique must be used which will not significantly alter the running characteristics of the system.

The required CPU cycles and I/O activity can be significantly reduced by using sampling techniques instead of employing event-oriented monitoring techniques. Using sampling, the CPU overhead, I/O activity, and secondary storage requirements are kept to a minimum as a result of the small volume of data collected. The accuracy of the sampling technique is based on the central limit theorem. The central limit theorem states that the accuracy achieved improves as the number of random samples increases. Figure 1 illustrates the confidence curves from which the accuracy of such data can be obtained, and illustrates that there is no substantial improvement in accuracy after approximately 26,000 samples.

The system interference caused by primary storage (core) requirements can be minimized by separating the extraction process from the analysis phase. This technique, along with reducing the core requirements, provides the ability to reanalyze the data collected since it must be placed in an intermediate storage area.

* Formerly of Boole & Babbage, Inc. Cupertino, California



CONFIDENCE CURVES FOR THE 99.99%, 99.9%, 99%, 95% AND 90% CONFIDENCE LEVELS

If 8,000 samples are taken, the probability that we will be within 2% of the actual value is 99.99%

Figure 1

These techniques and considerations resulted in the introduction of the first commercially-available computer performance measurement products.* The program evaluator (PPE^{TM}) provides insight into the distribution of activity within a program by isolating areas of high usage through a histogram report (Figure 2), which breaks down activity within overlay segments and individual program CSECTS, and pinpoints wait time caused by I/O. These data are provided by sampling the program under study in its normal operational mode by using a special extraction module.

The configuration evaluator (CUE^{TM}) provides the user with insight into the uses of his software-controlled hardware system. This is provided by supplying the user with resource utilization and queueing statistics, head movement data, transient SVC usage, and initiator/terminator activity.

In developing these products, it was recognized that regardless of the power provided by a commercially-available systems measurement tool, whether it be hardware or software, there are six essential considerations which a user must carefully consider before purchasing (or leasing) such a tool. These considerations are:

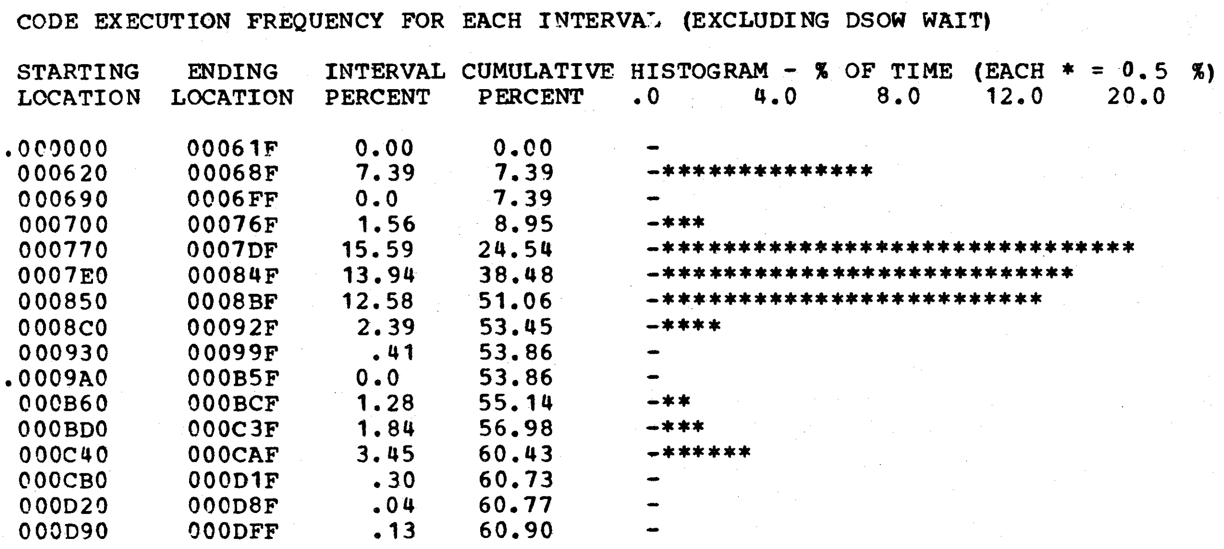
- maintenance provided
- life cycle
- documentation/training provided
- logical extensions to the tool
- ease of use of measures provided
- interrelationship with other tools

It is these considerations that molded the customer-vendor relationship that was adopted.

Since software measurement tools are dependent on the ability to extract from memory the necessary descriptive and quantitative data, the maintenance provided with a measurement tool is of the utmost importance. Since changes to an operating system can drastically affect such tools, the user must be assured that these tools will function properly as the operating system changes. At the same time he must be confident that the content of the data provided stays consistent with the evolutions of the operating system. In order to accomplish this five different extraction programs are maintained for the PPE product alone; three for the IBM 360/370 series, one for the SPECTRA line, and another for CDC customers running under master.

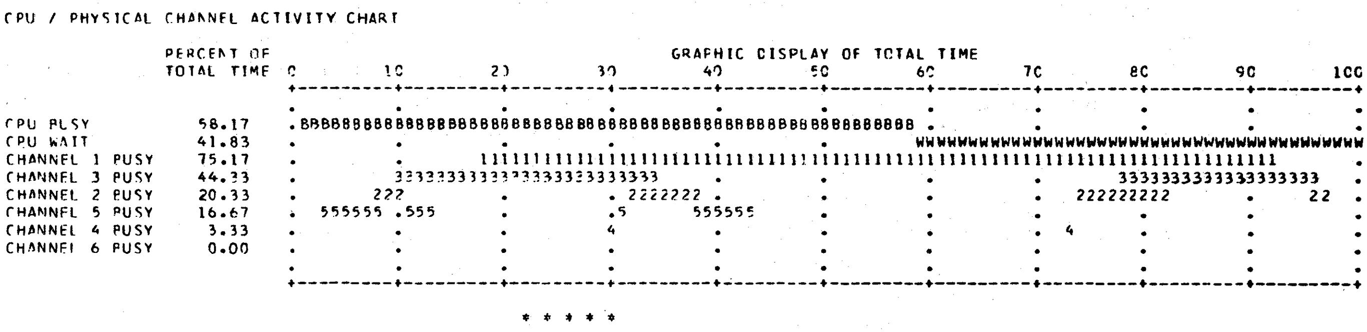
At the same time, products have been extended under standard maintenance agreements with the customer to reflect changes in both user and system philosophy. For example, the latest version of CUE provides existing customers with (1) graphic display of CPU and physical channel activity (Figure 3) where none was available before, (2) physical channel activity by logical channel where only physical channel activity was available before, (3) the ability to request the usage of any Boolean combination of these resources at analysis time instead of getting a fixed set of combinations, (4) direct access volume activity in sum-

* U.S. Patent Number 3,644,936



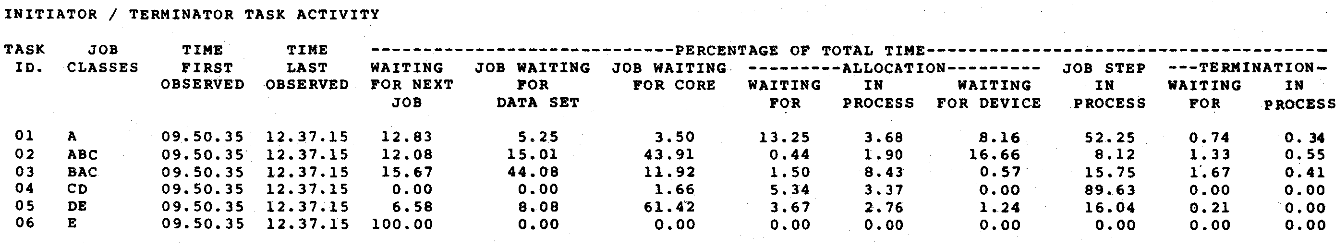
```
CODE EXECUTION FREQUENCY FOR EACH INTERVAL (EXCLUDING DSOW WAIT)

STARTING  ENDING    INTERVAL CUMULATIVE HISTOGRAM - % OF TIME (EACH * = 0.5 %)
LOCATION  LOCATION  PERCENT   PERCENT   .0      4.0     8.0     12.0     20.0

.000000   00061F     0.00      0.00     -
 000620   00068F     7.39      7.39     -***************
 000690   0006FF     0.0       7.39     -
 000700   00076F     1.56      8.95     -***
 000770   0007DF    15.59     24.54     -*******************************
 0007E0   00084F    13.94     38.48     -***************************
 000850   0008BF    12.58     51.06     -*************************
 0008C0   00092F     2.39     53.45     -****
 000930   00099F      .41     53.86     -
.0009A0   000B5F     0.0      53.86     -
 000B60   000BCF     1.28     55.14     -**
 000BD0   000C3F     1.84     56.98     -***
 000C40   000CAF     3.45     60.43     -******
 000CB0   000D1F      .30     60.73     -
 000D20   000D8F      .04     60.77     -
 000D90   000DFF      .13     60.90     -
```

Figure 2



```
CPU / PHYSICAL CHANNEL ACTIVITY CHART

                PERCENT OF                                           GRAPHIC CISPLAY OF TCTAL TIME
                TOTAL TIME    C         1C         2)         3)         4)         5C         6C         7C         8C         9C        1CC
                              +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+

                              .          .          .          .          .          .          .          .          .          .          .
CPU PLSY             58.17    .BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB .          .          .          .          .
CPU WAIT             41.83    .          .          .          .          .          .        WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
CHANNEL 1 PUSY       75.17    .          .        11111111111111111111111111111111111111111111111111111111111111111111111111111111111       .
CHANNEL 3 PUSY       44.33    .         33333333333333333333333333333     .          .          .          .       333333333333333333333    .
CHANNEL 2 BUSY       20.33    .      222            .          . 2222222  .          .          .          .  222222222          .     22   .
CHANNEL 5 PUSY       16.67    .   555555 .555       .          .5      555555        .          .          .          .          .          .
CHANNEL 4 PUSY        3.33    .          .          .          4          .          .          .          .  4       .          .          .
CHANNEI 6 PUSY        0.00    .          .          .          .          .          .          .          .          .          .          .
                              .          .          .          .          .          .          .          .          .          .          .
                              +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+

                                                          * * * * *
```

Figure 3



```
INITIATOR / TERMINATOR TASK ACTIVITY

TASK   JOB      TIME      TIME      ----------------------------PERCENTAGE OF TOTAL TIME------------------------------------
 ID.  CLASSES   FIRST     LAST      WAITING   JOB WAITING  JOB WAITING  --------ALLOCATION--------  JOB STEP   ---TERMINATION-
                OBSERVED  OBSERVED  FOR NEXT     FOR        FOR CORE   WAITING    IN     WAITING     IN      WAITING    IN
                                      JOB      DATA SET                  FOR    PROCESS FOR DEVICE PROCESS     FOR    PROCESS

 01   A         09.50.35  12.37.15   12.83      5.25          3.50     13.25     3.68      8.16     52.25     0.74     0. 34
 02   ABC       09.50.35  12.37.15   12.08     15.01         43.91      0.44     1.90     16.66      8.12     1.33     0.55
 03   BAC       09.50.35  12.37.15   15.67     44.08         11.92      1.50     8.43      0.57     15.75     1.67     0.41
 04   CD        09.50.35  12.37.15    0.00      0.00          1.66      5.34     3.37      0.00     89.63     0.00     0.00
 05   DE        09.50.35  12.37.15    6.58      8.08         61.42      3.67     2.76      1.24     16.04     0.21     0.00
 06   E         09.50.35  12.37.15  100.00      0.00          0.00      0.00     0.00      0.00      0.00     0.00     0.00
```

Figure 4

mary as well as detailed form instead of only detailed head movement by device, etc.

Once the user has established the serviceability of a measurement tool, he must evaluate its expected life. Is the operating system on which it is running expected to remain in the installation for a reasonable length of time? Does the supplier provide "trade-in" services for converting to a new operating system? Both of these questions should be carefully evaluated and the conclusions considered when calculating the true cost of the product.

There are many data processing tools which lack the proper documentation and user training necessary to reap their true potential. One of the biggest problems facing installations using measurement tools is to properly train and condition their data processing staff in the use and understanding of system measurement. Without proper documentation and training by the supplier, this understanding can never be achieved. The vendor should provide his customers with manuals describing the products and their use, personally install the product and work with the customer representatives until they become familiar with its use and, last, conduct a training course for all the installations' interested personnel.

Regardless of the type of package being used by an installation, a user always needs and/or desires some variation of the information provided in order to satisfy his needs. Consequently, he should be able to influence the expansion of a product line to fit his and other customers' needs instead of the vendor sitting in a well-insulated ivory tower dictating the measurement needs of the computing community. The problem which arises from this is how customers can formulate their ideas and communicate them to the measurement tool supplier. Two methods of customer-vendor communication which have been used by system measurement suppliers are to (1) help their customers establish users groups where customers can discuss their problems, expound on their successes and formulate their desires, and (2) constantly stay in touch with customers in order to get their feelings and establish their needs. These techniques have brought about several additions to the basic products, such as the addition of Initiator/Terminator usage reporting in CUE (Figure 4) and the PL/1 statement number option in PPE.

Many claims are constantly being made by vendors as to the power of their products, but it is extremely important for the user to determine the usefulness of the measures provided by these tools. Unless this usefulness can be established, the tool has relatively little value to the user. Consequently, it is desirable, from a user's point of view, that the reports be readable and not cluttered with useless information, and that each measure be carefully identified as to how it can be used in improving system performance.

A final consideration in choosing measurement tools is how the various tools, and even measures within a tool, relate to one another. If the results of the various measures can direct the user through his analysis phase, a much more comprehensive study can be performed. As a result, it is important that the various performance measures and products be inter-related to one another. It has been the goal to structure products such that all the tools in the System Measurement Systems (SMS) product line operate as a closely-knit group, Figure 5, (i.e., each product can direct the user to a further level of analysis that is provided in one or more of the other three products) and that each of their products be so arranged that there is a hierarchic structure (i.e., a logical flow of analysis) associated with each of them.

The establishment of a system measurement philosophy, such as outlined above, is one thing, but where is measurement going from here? Many people in the measurement field feel there is going to be an ever-greater emphasis on vendor-supplied software measurement products while the significance of hardware monitoring in the measurement field will decline. Hardware monitoring will probably be provided by the manufacturers in their equipment as diagnostic aids and will be used for first-level measurement (CPU and channel busy data, effects of cache memory, bulk core

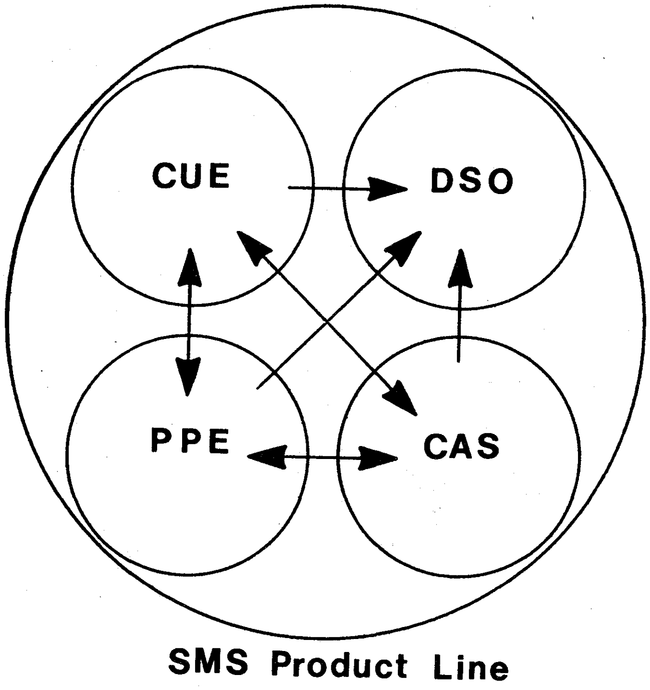


Figure 5

vs. main core, etc.). Software, on the other hand, will provide the second- and third-level measurements.

With the ever-increasing speed of CPUs and increased technology in channels (such as the block multiplexer), the emphasis is going to be directed away from these areas and toward I/O. This attention will be directed toward inter- and intra-data set contention.

What access methods are being used?
Was the proper data set organization used?
What effect did data set Y have on the device queueing?

Associated with I/O contention areas, effort will be directed toward the actual sources of system contention areas. What program caused the bottleneck and why? What system buffer area or table is too small or too large, etc.? This area in turn is going to direct emphasis toward program architecture. The organization of internal, as well as external, data areas and program flow will be stressed in an effort to increase total system performance. As a result of all these areas, programming and system design standards will develop and systems measurement will finally become the science it has been trying to achieve.

Historical perspectives—Computer Architecture

by MAURICE V. WILKES

University of Cambridge
Cambridge, England

INTRODUCTION

I shall attempt in this paper to record my own personal impressions of the way in which computer architecture and designers' objectives have changed during the period that has elapsed since the first stored program computers were being designed 25 years ago. I shall be concerned with the generally accepted "state of the art" at any time, rather than with the new ideas that were emerging in the more advanced centers. I shall not attempt to track innovations to their source nor to assign dates to them. For convenience of presentation, but for no other reason, I shall divide the period into three phases of very roughly equal duration. I shall end with some remarks about the way thinking has developed on the subject of memory hierarchies, this being one that can be traced through all three phases.

The very first automatic computers were not electronic nor did they have a stored program. The idea of a purely electronic large scale computer was first conceived and developed by J. Presper Eckert and John W. Mauchly at the Moore School of Electrical Engineering in Philadelphia. Their first computer, the ENIAC, showed very clearly the influence of mechanical ways of thinking. The accumulators, for example, contained rings of flip-flops, ten to each ring. At a given time one flip-flop was set and the others unset according to the digit being stored. The rings could be stepped from one condition to the next in much the same way that a mechanical counter wheel is stepped. Eckert and Mauchley came to realize that there were more efficient ways of approaching the design of an electronic computer than this and their work received a great stimulus when Von Neumann associated himself with their group. The ideas evolved were summarized by Von Neumann in a document entitled "Draft report on the EDVAC" which received some circulation during the early part of 1946. The material in this report and much else besides was presented to a small group of which I had the good fortune to be a member at the Moore School in the late summer of 1946. It may be said that the principles of the modern computer were then clear and that the events of the last 25 years have been their logical working out. However, not everyone recognized that this was the case and much energy had to be spent in countering the arguments of those who did not accept the stored-program principle or who had not sufficient faith that electronic technology would prove equal to the demands that would be made on it.

THE FIRST PHASE

The early stored program computers were designed to be operated by the programmer himself who could book the machine for a specified period, or stand in line waiting his turn with other programmers. By watching the machine as it was running he could learn a great deal about his program, particularly if it were not fully debugged, and one of the designer's objectives was to provide facilities for this purpose. Early memories, unlike the core memories introduced later, were dynamic in operation and lent themselves to the provision of means whereby the programmer could see what was going on in the memory while the machine was running. In the case of the ultrasonic memory, for example, it was easy to provide a cathode ray tube monitor that would show on a raster the pulses circulating in one of the tanks. By switching from tank to tank it was possible to examine any word in the memory. While the program was running some words would of course change too rapidly to be followed, but others would change more slowly, and it was interesting and instructive to sit in front of the computer and watch what was going on. Similarly, with the Williams tube memory, it was easy to provide a slave tube that could be switched in parallel with any one of the memory tubes.

By making use of the single shot button the progammer could follow what happened as each instruction was executed. Many computers had elaborate operating consoles by means of which the programmer could change words in memory as he debugged his program. Not everyone was happy about methods of debugging that depended heavily on the use of the single shot button, it being recognized that while they might be efficient for the human being they were inefficient for the machine. Even in this early period there was pressure toward having the machine run by professional operators and training programmers to use software—although that term was not then current—aids to debugging, such as traces, selective dumps of memory, post mortem routines that would print out words in memory that had been changed during the running of the program, and so on. However, the other method of working continued to have its advocates and in the case of some of the larger machines a whole group would move in and take over the machine when their turn came with an impressive display of bustle.

During this period designers of computers—one did not yet talk of computer systems—were struggling to liberate themselves from their own prejudices. There were also customers' prejudices to be considered to a much greater extent than is the case today. This was partly because of the general novelty of the ideas and the interest that they aroused among people at large, and partly because there was little experience to quote; often computers were sold off the drawing board before any working example was available for inspection and the first customer was at the same time the sponsor for the development work involved. Many of the prejudices related to the supposed differences between computers designed for business purposes and those designed for scientific purposes, and to the vexed question of binary versus decimal. If one believed that the distinction between business and scientific computers would cease to exist and that binary computers could be used quite successfully for business data processing, then it was as well that one should keep one's opinions to oneself, since to many people it seemed obvious that the contrary was true. Even the one distinction between business and scientific computers that appeared fairly clear proved to be a myth; this was that business computers would require faster and more efficient card readers and printers than would scientific computers. The truth was that in the early years all computers were crippled by their input and output devices and that when more efficient devices became available it was seen that they were wanted on the scientific side quite as much as on the business side.

Whenever computer designers got together there was much discussion about the logical design of the processor and the exact composition of the instruction set. Computer users were also concerned in these arguments, partly for the reasons mentioned above and partly because this was before the days of high level programming languages. One controversy concerned the number of addresses in an instruction. Three address codes had many advocates who claimed that compared with single address codes they led to more natural programming. With a three address code one could in one instruction cause two numbers from memory to be added together and the result returned to memory. In a three address computer in pure form nothing would survive in the arithmetic unit when the execution of an instruction had been completed. It always seemed to me that the argument for three address codes was emotional rather than scientific and I was not surprised when eventually they disappeared.

All the early computers had multipliers but some dispensed with dividers in order to reduce the amount of logic required; it must be remembered that we had only vacuum tubes and not yet much practice in using them in digital circuits. The high cost, large physical bulk, and poor reliability of vacuum tube logic lay behind every argument, whether it concerned the mode of operation—serial or parallel—the provision of checking circuits, or the complexity of the instruction set. Always the argument for simplicity was a strong one. An example is floating point arithmetic. Although floating point arithmetic had been provided in early relay computers, we fought shy of it in electronic computers for a long time because of the complex logic involved. We used to whistle to keep up our courage and would assert loudly that floating point instructions were unnecessary since one could use a subroutine. In spite of this, the later vacuum tube computers had floating point units. Oddly enough, some numerical analysts opposed the introduction of floating point arithmetic on the grounds that floating point operation was treacherous to the unwary and that programmers would get themselves into trouble by misusing it. Although dividers became common, very few stored program computers had hardware units for extracting a square root, in spite of the fact that this had seemed to the designers of the ENIAC a necessary feature. Von Neumann told me in the summer of 1949 that he thought it possible that the hardware extraction of square roots would come back on account of the importance of the square root in Euclidean geometry.

Fundamental matters that were much discussed were the number of bits in a word and the number of instructions per word. These have, of course, remained important considerations. The need to be able to handle groups of bits, or even individual bits, in a word was recognized, and indeed some quite early instruction sets had provided means for selecting groups of digits and shifting them into the required position in one operation. This need led eventually to the development of byte organized machines. This move was by no means solely a response to the needs of business data processing. The area remains one in which there is a good deal of diversity in the solutions that individual designers adopt. In view of the fundamental nature of the considerations involved I believe that this situation will continue. In the early years there was not yet any heavy investment in computers and the need for compatability had not begun to rear its ugly head. Designers were, therefore, more free to adopt what appeared to them to be optimum solutions. While the need for preserving compatability will remain a major consideration in the short term, I believe that its long-term influence will be less than some people think.

Early computers consisted of a number of units, all different; it took a few years before rational ways of packaging a computer were discovered that would increase the proportion of repeated units. I gave a talk at a conference at Manchester University held in 1951 with the title "The best way to design an automatic calculating machine". This paper has been frequently referenced because it introduced the idea of microprogramming and attempted thereby to bring some order into the design of the control section of a computer. The paper was, however, also concerned with the problem of introducing as much repetition as possible into the construction of the registers and adders, both control and arithmetic. I was thinking then in terms of fairly large replicated units. A year or two later people began to develop small plug-in units, each containing a small number of logical elements. These could be used in many combinations, it being accepted that some would contain wasted components. This step was not as easy as may appear since the packages had to be designed to rigorous standards so that they would be interchangeable. In the very early days we could not have done this. The advance may be compared with that made in mechanical engineering when interchangeable components were introduced. The coming of integrated circuits has, of course, reopened the whole question of how a computer should be packaged.

THE SECOND PHASE

As computers got bigger we saw the introduction of batch processing. A computer was fed with a magnetic tape on which was written a batch of jobs in a certain order. Eventually the computer produced an output tape containing the results in the same order. A separate auxiliary computer was used for the initial batching of the jobs on to the input tape and for printing the results from the output tape. This computer with its tape decks was usually quite separate from the main computer. An essential feature of batch processing was that jobs went through their various stages of compiling, assembling, and running as a batch, intermediate tapes being written as required. There was no possibility of a short job overtaking a long one, nor was there any possibility of a programmer being able to watch his job being run.

From the system point of view the importance of batch processing was that it went far toward solving a problem that had troubled computer designers from the beginning, namely how to secure efficient cooperation between a processor and its peripherals. In early computers the processor was mostly idle when either the printer or the card reader was in action. It is true that the design of the hardware usually allowed the program to continue after an output order had set the printer in action or an input order had caused the feeding of a new card. In practice, however, this facility was of little value since, during the operation of an input or an output subroutine, the only computation available to be done between consecutive input and output orders was conversion to or from the binary scale. Even in cases where useful overlapping of operation of the processor and of the peripherals was possible, the programmer had to go to quite a lot of trouble in order to achieve it. It had early been realized that the use of magnetic tape for input and output would be one way of tackling this problem. However, off-line equipment had then to be provided for writing the input tapes and printing the results from the output tapes. Attempts to meet this need met with only partial success until the adoption for the purpose of the auxiliary computer.

It is interesting to note that the term "batch processing" has survived the systems to which it was aptly applied. The term is now used in contradistinction to the interactive mode of operation. Jobs commonly flow through the system continuously rather than in batches, and very often there is provision for jobs of high priority to overtake jobs of lower priority.

Batch processing required a supervisor or monitor, part of which was permanently resident in core. It also required sufficient magnetic tape decks for the input and output tapes to be permanently loaded along with a system tape, while at the same time leaving enough tape decks available for loading, preferably in advance of their actually being required, the magnetic tapes required by individual problems.

Batch processing was made possible by the fact that computers were getting larger and better endowed with peripherals such as magnetic tape decks. It was the increase in size of core memories, however, that was decisive. This was a time when a memory as large as 8k words was unusual, and 16k, to say nothing of 32k, seemed a lot. The availability of large core memories also made compilers possible and one remarkable feature of the period was the way in which users appreciated very rapidly the advantages of programming in high level languages—automatic programming, as it was called—and were willing to pay for the large core memories required.

Computers had now become large and powerful enough, and their reliability was sufficiently high, for large-scale computer applications to be attempted. The importance of software began to be appreciated and the foundations were laid for the modern development of that subject. There was less public discussion of the details of computer design, but much was going on behind the scenes. It was a truly traumatic period for the circuit engineer who was having to come to grips with the use of transistors instead of vacuum tubes. This meant learning his trade all over again. This was all the more difficult since in the early days transistors were slow and unreliable devices and were by no means ideal elements from which to build digital circuits. Within a few years they had been developed to a point at which they enabled better standards of performance to be achieved than would ever have been possible with vacuum tubes.

A number of large-scale projects were established in this period and several of these, in particular STRETCH and LARK, contained the germs of much later development in computer system design. It was about this time, in fact, that people first began to talk about computer *systems*. In the early days the processor had been the most prominent part of the computer and it came provided with a fixed amount of memory and a more-or-less standard set of peripherals. Nowadays, we regard processors, memory units, tape decks, drums, disc files, printers, and card readers as blocks from which to build a configuration suited to a particular user's needs. We may have more than one processor and we may have computers, perhaps on remote sites, connected together to form an integrated system. We still tend, however, to think of the processor as being a major item and a range of computers as being defined by a range of processors. It remains to be seen whether we shall always do this.

THE THIRD PHASE

The development of the concept of an operating system was a great advance but it was really an advance on the software front. The operating system took over some of the duties of the operators, but otherwise the operation of the computer was very much as it had always been. The next advance came when it began to be taken for granted that a computer should have an interrupt system.

In most early computers a break in the flow of control could only occur as the result of the execution of a jump instruction. An unscheduled event, such as an accumulator overflow, if detected at all, would most likely cause the computer to stop. Such unscheduled events are now often described as traps and lead to a jump in control. Interrupts are essentially traps coming from outside, usually from a peripheral device that requires attention. In a small computer, a very simple interrupt system may be all that is required, but in larger systems the hardware must detect and store interrupts coming from a number of sources, and it must service them according to a priority partly wired into the hardware and partly determined by software. There must be the ability to postpone the servicing of interrupts at critical moments, for example, while earlier interrupts are being serviced. Altogether, the need to provide an effective interrupt system has added substantially to the hardware designer's cares.

The provision of a system of interrupts enabled a processor to interact efficiently with mechanical devices which now included disc files as well as tape decks and ordinary input or output devices. It became possible to use buffered input and output so that card readers and line printers could run independently of the main computation. Multiprogramming became possible; this implied that there should be resident in core programs for two or more independent jobs so that when one was held up, hopefully, another would be free to run. These developments would not have been possible unless sufficiently large core memories had become available at the right time.

Interrupts, as I have said, made it possible for the processor to interact efficiently with its slow peripherals. It was in due course realized that the peripherals need not be ordinary computer peripherals but could be, for example, control valves or other devices in an industrial plant. There was some excitement when the possibilities thus opened up in control engineering and related fields were fully realized. The term "time-sharing" was first used in this connection. Later, when the same techniques were applied to enable a group of users sitting at consoles to make independent use of a computer, the word was preempted and has now come to mean that alone.

Another need for which the computer designer was now expected to make provision was that of memory protection. This need had been felt—although not at once satisfied—as soon as batch processing systems with a permanently resident monitor came into use. Without memory protection a user's program that contained, by inadvertence or otherwise, an instruction that corrupted the monitor could cause a system to break down. The development of time-sharing made the need for memory protection imperative.

At first, systems in which protection was all-on or all-off were thought to be sufficient. It was then possible to write the supervisor in such a way that it could not be interfered with by a user program, and so that one user program could not interfere with another user program. This is quite satisfactory if the supervisor is completely debugged and if no changes—with the inevitable hazard of bugs—are made to it. It suffers, however, from the disadvantage that use of the facilities for protection is the sole prerogative of the writer of the supervisor and it is not possible for a user to construct a sub-system whose parts are protected from one another. Sub-systems requiring such protection must be incorporated wholly or partly into the supervisor with the hazards just referred to. The desire to have operating systems that are robust against hardware and software failure is leading to a demand for systems of protection whereby all routines, whether they belong to the supervisor, to a sub-system, or to the program of an ordinary user, can be surrounded by protecting barriers. Under normal circumstances, these barriers are redundant, but when accidents occur they act like fire walls and prevent the damage from spreading. Hierarchic protection systems based on rings or levels of protection are now beginning to appear and advanced research is being done on non-hierarchic systems.

The effect of these developments has been materially to broaden our view of what comprises a computer system. No longer is it a simple computer with peripherals connected to it. In addition to the main computer there will probably be auxiliary computers used for a variety of purposes such as communications, for providing user support services, for controlling graphic display devices or plotters, and so on. Some of these will be adjacent to the main computer and some may be situated remotely. These computers are to be regarded as satellites to the main computer and as being hierarchically dependent on it. The main computer may, in addition, have core to core links to other computers with which it communicates on equal terms for the purposes of load-sharing or exchange of data. The whole system may be connected to a context free network of the ARPA net type through which communication with other systems small and large can be effected.

Along with the interest in computer systems has gone a welcome revival of public discussion of the processor and what it can do. This is shown very clearly in the interest in microprogramming that has grown up during the last ten years. A microprogrammable processor, "on a chip", may well come to be one of the basic elements from which computer systems are constructed.

MEMORY HIERARCHIES

The early years of computer development were taken up with the quest for a satisfactory form of high speed memory that would be reliable and would hold out the hope of large memories becoming possible. Many things were suggested and many were tried; the only forms of memory that had any success in practice were those based on the use of mercury tanks or cathode ray tubes. The coming of the core memory changed all this. Magnetic drums and tapes were spoken about from the beginning; oddly enough, so was magnetic wire in spite of the practical difficulties involved and of the fact that it could only accommodate one track. Disc files first appeared in the second phase mentioned above, but it was some years before they came into general use.

At first designers took it for granted that it was the responsibility of the programmer to move material between the auxiliary and the high speed memory. It was not until the second phase that the possibility was entertained that automatic transfer might be possible, so that the two levels of memory would appear as one to the programmer. It is, however, only in recent years that paging has been taken up on a wide scale and the design problems that it poses, particularly those of a software nature, are only just beginning to be understood. Meanwhile another idea had been introduced, namely the insertion of a super speed memory—known as a buffer or a cache—between the main high speed memory and the processor. This accumulates to itself the most recently read words from the high speed memory and delivers them up when they are next required with greater speed than would be possible if they had to be fetched again from the high speed memory. Philosophically, the idea is very similar to that of paging, but the practical designer finds little in common between them since the parameters are very different. The efficiency of a buffer in increasing the apparent speed of the high speed memory has made designers, even of very powerful computers, look toward core memory of economic speed rather than of the fastest speed obtainable. It will be interesting to see how the coming of semi-conductor memories affects these calculations. A further question of interest is whether the mapping of the high speed memory on to the buffer should be done in physical memory space or in virtual memory space. If the latter course is adopted, the time taken to add the content of a relocation register to an address is effectively reduced by the action of the buffer in the same proportion as the high speed memory access time. The use of the technique known as *pipe-lining* has been widely adopted for very fast processors and it will be interesting also to see how this is affected by the introduction of buffers.

ENVOI

Time-sharing has not had the impact that it should have had. This, I think, is partly because the emphasis was put on the high degree of interaction required for artificial intelligence, computer aided learning, and such subjects, and not sufficiently on the provision of facilities, such as convenient editing and running of programs, that are needed by the ordinary user. I feel, however, that the major portion of the blame must go to the industry, which was so busy designing systems intended to meet the requirements of the past that it failed to see what the requirements of the future would be. Although computer hardware has steadily progressed, operating systems are by and large anything up to 5 years behind the best experimental practice. Fifteen years ago it was apparent that the hardware designer and production engineer occupied dominating positions in a computer company. The result was failure to react to the needs of the programmer. One would have thought that, with the recognition of the commercial importance of software, this situation would have changed, but I do not think that it has. Computer companies now spend a great deal of money on software, it is true, but they treat it like another sort of hardware. When I first heard the term "software engineering" it seemed to me to be an excellent one, stressing as it did the need for a businesslike approach to the construction of software. I am afraid, however, that it was coined by people who thought that they could apply the methods of design and construction used in hardware engineering to software. The results have not been very good and it is easy to see why. The hierarchic structure of organization in hardware works because as the young engineer becomes more experienced and senior he can use his skill and experience in supervising others. He can do this because he can see what they are doing, and it is not too complicated for him to evaluate. It is not possible to supervise a team of software writers in the way that it is possible to supervise a team of engineers. Attempts to do so cause a lot of money to be spent and an article full of overheads to be produced. What sometimes saves the situation is that the young men at the bottom get together and go their own way.

There must be a hardware/software interface that is almost hardware and must be maintained by the same team that maintains the hardware. But this should be very small and certainly should not be a whole operating system. Operating systems are best developed in a user environment and not in the ivory tower inhabited by a computer manufacturer. I feel that what we badly need is a period of more open development of operating systems in which the benefits of competition can make themselves felt.

Historical perspectives on computers—Components

by J. H. POMERENE

IBM Corporation
Armonk, New York

INTRODUCTION

The technological base for computers was laid just prior to and during World War II. Military requirements led to a general upgrading of components. Television and radar pushed the development of high performance vacuum tubes, particularly the twin triode. Radar shifted the attention of engineers from frequency to time and timing. The microsecond became familiar.

Against this background a singular project began in 1943: An electronic calculator called ENIAC, planned to use almost 20,000 vacuum tubes. This was an unprecendented number, three orders of magnitude greater than state-of-the-art electronic products and ten times the size of anything else being considered. There was no assurance from past experience that the machine could ever work. Some observers predicted a tube failure every few seconds.

As it turned out there were only two or three tube failures per week and ENIAC was very productive. Actual experience was five orders of magnitude better than the worst predictions. The difference was due to design rules which minimized the probable causes of failure and to a less rigorous principle which says that reliability is often better than careful calculations show. Whatever—ENIAC established that large vacuum tube systems would work.

The ENIAC was primarily intended to compute ballistic tables. It was designed by analogy to electromechanical calculators and like them used decimal arithmetic and handled the digits of a number in parallel. Operations were timed by a 100 KHZ clock; addition time was 200 microseconds and multiplication required 2.8 milliseconds. Internal memory was very limited and consisted of 20 numbers of ten decimal digits held in vacuum tube accumulators. Instructions were not stored in memory; the machine was set up for each problem by means of pluggable wiring.

The speed of ENIAC attracted interest in solving problems outside of ballistics—hydrodynamics calculations, for example. Some of this was done but the limited memory and the manual labor of re-programming were severe limitations. It was recognized that a considerably larger memory ought to be provided for the basic data of the computation and that the same memory could also store the program to be followed. A useful capacity was estimated to be 4,000 words (i.e., either numbers or instructions). From this point forward memory technology was to dominate system design.[1,2]

EARLY MEMORIES

The choices for a 4,000-word memory in 1945 were not many. A word size on the order of 40 bits was required so that 160,000 total bits would be needed. Although a memory could in principle be made from the logic technology (i.e., vacuum tubes) this would require perhaps four tubes per bit or a total of 640,000—rather too many. It made better engineering sense to look for bulk memory effects, ways to store a large number of bits in one physical device.

One possibility was to launch pulses representing bits along a path having a large propagation delay, receive the pulses at the far end then re-amplify them and re-launch them. The memory would be the number of pulses in transit through the delay. Acoustic delay lines had been developed for radar with delays on the order of a millisecond and capable of transmitting one microsecond pulses, with consequent memory capacities on the order of 1000 bits per line. This kind of memory was used in one immediate successor to the ENIAC which was called the EDVAC. Since the memory was inherently serial EDVAC was organized on a serial basis, that is the bits of a number were handled in time succession rather than all at the same time.

Another possibility was very obvious in principle: the iconoscope which could store and retrieve a television picture with over 200,000 resolvable elements. For rigorous digital storage this capacity would have to be much derated, though by how much was not known. The basic attraction of the approach was that an electron beam could be used to lay down a pattern of bits on a suitable surface at very high densities and that any of these bits could be selected for retrieval by the same beam. This random, rather than serial, accessibility to bits permitted (though did not require) parallel handling of a number. A second immediate successor to the ENIAC was built, using this parallel approach, at the Institute for Advanced Study.

Although suggesting the approach the iconoscope itself was not used for electrostatic memory. Several alternates were suggested or tried but the most widely used system was based on a development by F. C. Williams using ordinary cathode ray tubes. This scheme was based on small area differences in secondary electron redistribution and may be one of the few utilizations of a third-order effect.

It was possible to store 1024 bits per tube without getting unacceptable coupling between adjacent charge distributions. The worst case of this coupling, picturesquely called spill, occurred when one bit was read many times more than its immediate neighbors. All electrostatic memories required some explicit or implicit measures to control spill.

Delay line and electrostatic memories provided capacities from 512 to 2048 words, short of the estimated 4000 but enough to get computing well started.[3, 4]

EARLY LOGIC AND PACKAGING

Basic considerations

It was clear that at least the following points had to be considered in designing a digital computer:

(a) Large numbers of tubes and other components would be used, all of which would have to work very reliably.
(b) Electrical signals representing the numbers would have to be kept within operable limits throughout the machine and over long periods of time.
(c) Since all possible bit patterns could occur within the machine the DC component of signals would have to be taken into account.

These overall considerations entered into many of the more specific decisions.

Derating and tolerances

The major components to be used were vacuum tubes, composition resistors, and in some machines pulse transformers and crystal diodes also. Tubes could fail catastrophically by heater failure or internal shorts and gradually through loss of cathode emission. Resistors could drift away from initial values. Diodes probably did not wear out but could be destroyed by overload of very short duration.

Heater failures were controlled in the ENIAC by never turning the heater off, so that thermal cycling did not occur. IAS designers felt that most heater damage came from the shock of rapid heating or cooling and provided for gradual turn on and off. Both techniques worked very well. Internal shorts were minimized by rigid pretesting including vibration, choice of tube types, and careful control of heater—cathode potentials. Emission loss was allowed for by designing circuits which would still work at half the nominal operating current. Power dissipation was derated usually to 50 percent and sometimes more.

Resistor deterioration was known to be accelerated by high power dissipation so all were derated to half power. Initial values were held to a 5 percent tolerance, or better in critical circuits, but all circuits were designed to be operable with all resistors off nominal by 10 percent in the worst direction.

Diodes were protected by designing circuits in which overload conditions could not occur or at least required unlikely failures in other components.

Vacuum tube choices

The basic logical "AND" operation was provided in two ways. With a multigrid tube the control grid and the suppressor grid were used. With triodes two or more were used together either with common plates or common cathodes. The triode circuit was easily extended to more than two inputs and was widely used. It was common to employ the "long tailed pair" in which the cathodes were connected together and returned to a negative voltage large compared to the grid-cathode voltage, typically—150 or—300 volts. In this arrangement the tube current was largely determined by the cathode resistor and the return voltage; an important technique to minimize effects of tube deterioration.

Unfortunately the signal out of a vacuum tube circuit is always at an appreciably more positive level than its input and must be translated back down to be used as an input to the next circuit. The capacitor coupling common in communication circuits could not in general be used because it blocked the DC component of the signal. DC translation required a resistor divider returned to a large enough negative voltage to minimize signal attenuation.

This translation network was a major limitation on speed. In order to have an adequate output signal under worst case tolerance it was necessary to have a large nominal signal swing at the plate. Since tube current was limited by derating, the network impedance had to be pushed up and circuit delay along with it. As crystal diodes became available with ratings compatible with tube signal swings (e.g., 30 volts) this problem led most designers to use tubes mainly for re-amplification and powering and to do the logical operations with diodes.

Pulses vs. DC coupling

One way to handle the DC component of signals was to use direct coupling, as described above. This allows the system to run at any speed up to its maximum which need not be known in advance of design. Another way was to represent information bits by pulses and restore the DC component when necessary by clamping diodes. Adequate DC restoration required some advance decisions affecting final speed, such as a standard pulse width. This difference afforded some interesting debate but final speeds turned out about the same.

Pulse logic systems were timed with explicit clock signals. Though most direct coupled systems were also clocked the timing latitude available permitted an asynchronous mode of operation. This mode had the advantage, in principle, of being insensitive to timing changes—as tubes degraded the machine would still run albeit more slowly. One form of asynchronism was used in the IAS machine. Timing was determined by circuits analogous to those being controlled, and affected in the same way by supply voltages and control wave forms. This compensated for several kinds of deterioration which could occur but not all.

A more complete concept of asynchronism was self-timing logic. In one version a signal would be propagated separately in both true and complement form, the arrival of one or another at the end of the chain would signal completion. The Philco S-2000 embodied logic of this kind.

Wiring and connections

Wiring presented no problems unique to computers. Layout tended to be generally planar but all three dimensions were used for wire routing. Wiring impedance was largely uncontrolled, except for being kept as high as possible (i.e., a thin wire in free space) in some direct coupled machines. Even so, circuit impedances were higher yet and capacitive loading was a problem. Pulse logic machines represented a low enough source impedance to largely avoid capacitance problems.

The tubes and circuitry associated with a logical grouping, such as a register position, were generally packaged together as a plug-in unit. Layout within the unit was three-dimensional and followed signal flow where possible to minimize connection lengths. Signals going outside the unit were driven by cathode followers.

Two non-commercial machines, the SEAC and the IAS, probably represented the packaging extremes. Signals in the SEAC were driven at low impedance from pulse transformers and wiring could be any reasonable length along any route. As a result a convenient rack and panel construction was used, logic and crystal diodes on the outside for accessibility and hot tubes and resistors on the inside. Wiring going any length ran in longitudinal trays.

The IAS machine was direct coupled and hence high impedance. To minimize capacitance loading the physical layout followed the logic flow very closely so that all signal wires were very short. Chassis were curved so that all intra-chassis wiring could be point-to-point away from the chassis (like chords of a circle).

Power and cooling

The heater of a computer tube required between 0.45 and 0.9 amperes, or 450 to 900 amperes per 1000 tubes. Supplying and distributing such large currents presented no basic difficulty but heaters were always a nuisance and certainly not less so in large numbers. DC power was more of a problem. Loads aggregating to 30 amperes at 300 volts were common and similarly for other voltages. Commercial supplies of this size were not initially available, so early groups had to plan their own. In some cases large storage battery banks were floated across a DC generator. This had some advantage during construction, when voltage levels and loads were subject to change, but it was not operationally convenient. Subsequently, very satisfactory commercial supplies were produced. These used thyratrons in a 3-phase full wave circuit and had filtering capacitance of nearly a farad. Regulation was easily held to a few percent.

Cooling was done with forced chilled air. The only real difficulty, at least for the earlier machines, was learning how to operate air conditioning in the winter.

FERRITE CORE MEMORIES

Origin

As computers became operational and were put into use attention turned to increasing the memory capacity. For the serial delay line machines an auxiliary magnetic drum memory could offer considerably increased capacity at nearly the speed of the delay line memory. For the faster parallel electrostatic machines the same kind of drum was too slow to be other than a secondary memory.

At about this time (1951) the ferrite core memory was developed at MIT and at RCA. The MIT memory replaced the electrostatic system on Whirlwind and provided 2048 words; the RCA memory was a 10,000-bit plane. At the outset the ferrite memory offered speeds equal to or better than electrostatic and capacities appreciably greater. Further, it seemed likely that speed and capacity could be increased by an order of magnitude with further development.

General characteristics

By earlier standards the ferrite memory was unlikely. Instead of storing 1000 bits in one delay line or 1024 bits in one cathode ray tube it required a ferrite core laboriously threaded with 3 or 4 wires for each and every bit. The assembly labor for one early core plane was 240 hours. Probably only a firm conviction that production could be automated gave courage to proceed.

Apart from the labor of core-stringing the prognosis for ferrite looked very good. The magnetic properties were a bulk effect susceptible to tight control and having no likely time deterioration. Although the drive circuits were fairly expensive it was expected that much larger arrays could be handled. Since the drive cost for a $n \times n$ array was proportional to n and the capacity was n^2 the drive cost per bit would be proportional to $1/n$. This is a strong economy of scale.

Manufacturing improvements

The investment in manufacturing automation depends on the expected volume of production. The great success of the ferrite memory came from the expectation by manufacturers that the volume would be very large. The first impetus came at MIT shortly after the development of the ferrite memory. In planning for the SAGE air defense system MIT worked out some basic techniques for core plane fabrication. IBM continued the work, both for SAGE and its own commercial production. IBM made a large and continuing investment in automated fabrication and also in automated pretesting of cores. The story of this has been well told elsewhere. The outcome was that memory changed from something rather special and difficult to something that was commonplace, easy to use, and could be as large as need and purse allowed.[5]

Speed improvements

The first ferrite memories used cores of a size that could be easily seen and through which wires could obviously be pushed. The resulting speeds were equal to or better than electrostatic memories and quite compatible with vacuum tube computers. For example, the IBM 701 with a 12 microsecond electrostatic memory was replaced by an improved IBM 704 with a 12 microsecond core memory, both matching the cycle of the arithmetic logic. However, the switch from vacuum tubes to transistors made arithmetic logic cycles of 0.3 microseconds seem possible: an improvement of 40 to 1. Note that this came from a change of kind in logic technology, not just degree. (Ferrite memories could never produce a change of degree to match this change of kind.)

Increasing the speed of a ferrite memory involves a number of factors but in any case the core size must go down. From early cores about the size of an aspirin pill the size went down toward the almost invisible. Problems of handling, threading and testing were solved on the way down but a barrier of sorts was reached at a cycle time of 0.5-1.0 microseconds. These were produced in quantity but the next step, seen as 250 nanoseconds, would involve a massive tooling effort to be practical. Semiconductor memory technology had meanwhile moved to where it could predictably offer higher speed and lower cost. The next ferrite step was not implemented.

THE ROLE OF PROGRAMMING

It took special skill, motivation, and patience to program the early machines and it was not obvious that use of computers would ever spread beyond a limited number of places where such expertise could beassembled. Attempts were soon made to use the computer itself to handle some of the labor of programming but the small capacity of the early memories limited what could be done. The larger ferrite memories removed this limitation and effective programming aids began to appear. Among these were programming languages which in effect re-defined the hardware computer into a new computing system. This new system could be programmed by a user in terms familiar to his discipline, he did not have to learn or deal with the intricacies of the hardware system.

One of the first languages was FORTRAN, which provided a computing system particularly easy for scientists and engineers to learn and use. As a result the entire technical community became actual or potential programmers and usage of computers skyrocketed.

This expansion of computing services to ever-widening circles of users became the driving force for the growth of the industry.

DISCRETE TRANSISTORS

The invention of the transistor attracted the immediate attention of computer designers. They were smaller and effectively faster than tubes and required no heater power. Work on transistor computers began as soon as enough transistors could be obtained and with the explosive growth of computing already apparent there was an urgency which produced rapid improvement. In short order it became possible to design machines which would be ten times faster than existing vacuum tube systems. The pace of improvement was still continuing, however, and even higher speeds seemed possible.

The STRETCH project of IBM is a well-documented example of this point in computer history. STRETCH designers set a goal to be one hundred times faster than the IBM 704, a goal to be reached by pushing hard on both component technology and logical machine organization. Though STRETCH did not meet the goal in all respects, the effort did result in a significant upgrading in all areas of computer technology.[6]

Circuits and logic

Not only did transistors have no heaters, they were also available in two complementary forms working on opposite voltages. The problem of signal voltage translation which had been such a significant limitation in direct coupled vacuum tube circuits could be handled very nicely by alternating the two kinds of transistors. The DC signal shift of one kind was compensated by an opposite shift in the other. All transistor machines used direct coupling.

The problem of saturation in transistors was avoided by the development of a circuit in which the saturation condition would not occur. This was called a current mode circuit and it was the transistor analog of the "long-tailed pair." This circuit was used in STRETCH and it is still widely used today where speed is important. Used with drift transistors it provided a circuit family with an average delay less than 20 nanoseconds.

The transistors, resistors, and diodes comprising a basic logic circuit were mounted on a small card with printed wiring interconnections. In many cases larger cards were also used to provide larger functional groupings having recurrent use. These larger cards carried on the order of 20 transistors. As with vacuum tubes these circuits were designed on a "worst-case" basis.

Memory limitations

Early core memories were driven with vacuum tube circuits. As part of the work on transistor machines development of a fast core memory with all transistor drive circuits was begun. This was a bold effort because it sought a threefold increase in speed over the best memory then available while accepting the handicap (at that time) of not using tube drivers. The goal was 2.0 microseconds and in fact a cycle of 2.18 microseconds was achieved. This was probably the last memory specified on a rigorous worst-case basis. In a way reminiscent of the spill problem in electrostatic memories it was observed that repeated accessing of the same location at maximum memory rate would result in heating the affected cores beyond the Curie point and result in loss of information. For this reason the array was cooled in oil. Somewhat later, on the realization that the worst case was exceedingly improbable, air cooling was substituted.[7]

Ambitious though it was, the two microsecond cycle fell far short of matching transistor speeds. In STRETCH, for example, the logic cycle was 300 nanoseconds, making the memory cycle seven times greater. In order to offset the speed imbalance the concept of lookahead was introduced. The memory would be kept as busy as possible supplying the next few instructions and operands in anticipation of their use. Unfortunately the critical importance of the branch instruction was not fully recognized. At a branch the program may take one of two paths and if the lookahead had gone down the wrong path considerable unwinding was necessary. This problem proved to be quite fundamental and had a strong effect on high performance machine organization.

Wiring and connections

The first level of wiring was now handled by printed wiring on the circuit cards. These cards then plugged into sockets on the back panel. Wire wrap was used rather than soldering, a choice which facilitated the widespread use of automated back panel wiring. Coaxial cable was used for critical leads and ordinary wire for the rest. This wire, though not really controlled in impedance, tended to be about 150 ohms in the back panel environment.

Although transistors took much less space than tubes they were also used more lavishly. As a result most computers were still too large to fit in one conveniently sized frame. Cabling between frames was conventional but on large machines like STRETCH it could also be described as monumental. Again, despite the much lower power consumption of transistors, their large numbers resulted in power and cooling requirements not much different from vacuum tube machines.

LARGER MEMORIES

Computers had grown along two diverging lines, scientific and commercial, and it was becoming apparent that this divergence was serving neither area well. Almost every installation really needed to do both kinds of work. A unification was needed but it could not be achieved without fundamental machine instruction and instruction format changes which could invalidate a majority of existing programs. Allowing the divergence to continue, however, would only let the problem grow larger. In announcing System/360 IBM opted for unification and set about to do what was technically possible to ease the transition.

Actually a more pedestrian force than the logical necessity of unification would probably have forced the same outcome. The 704-709-7090 had an address field of 15 bits, enough to address some 32,000 words of memory. Though apparently an ample allowance when the 704 was designed it had become evident that much more memory was needed to support the kind of programming service then in demand. However, the simple change of the address field to provide more bits would have caused most of the disruptive effects of the more comprehensive changes of System/360.

From our component-oriented standpoint the net effect was that much larger memory could be addressed (up to 16 million bytes or approximately 4 million words.) Once again this had a direct effect on system software, making possible a comprehensive operating system which in turn increased the utility of computers and stimulated further growth of the field.

Computer designers from the outset thought of the speed of light as a mere 1000 feet per microsecond. This was not a limitation in early machines but designers felt that it would become one. When the nanosecond speeds became possible the speed of light was then regarded as one foot per nanosecond and the limitation was more tangible. That which everyone knew was coming was suddenly at hand. The situation with integrated circuits is quite similar and also relevant to the speed question. As one looked at the physics underlying the semiconductor art one realized that there was no near-term limit to how small transistors, and their interconnections, could be made. The piece of silicon which once provided a single transistor could be made to hold a complete circuit of many transistors—an integrated circuit. The scale of this integration could be projected as quite large, and large scale integration (LSI) became the same sort of round-the-corner thing as speed of light limitations.

When it is precisely known what to make LSI seems promising indeed. But when there is less certainty and changes may have to be made after fabrication LSI becomes a problem. It is what it is and if even one detail is wrong it must either be accepted or thrown away.

The problem of changeability is increased when many different products might be brought out at about the same time. Partly for this reason IBM chose a hybrid approach to integration for System/360 production. Though the silicon chips were not initially integrated the entire manufacturing process was highly automated, affording many cost advantages. Integration of the chips was subsequently increased, moving nearer to LSI.

The IBM approach was transitional but it elaborated LSI technology. Circuit modules were mounted on boards with multiple layers of printed wiring. The characteristic impedance of this wiring was controlled at two levels: distributed transmission line runs and transmission lines lumped with successive circuit loads. Wiring between boards was conventional.

Control memory appeared as a new component in some systems, implementing the earlier idea of microprogramming. In this concept the regular machine instructions are themselves programmed from very elementary hardware operations. This eliminated the need for quite a lot of wired-in logic and illustrated one way of trading memory bits for logic circuits.

The first control memories were physically but not electrically changeable so that their contents would not be lost when power was shut off. The physical change required preparation of a new pattern corresponding to the new information content but this was still much easier than changing hard-wired logic. These memories made it feasible to alter the whole nature of a machine's instruction set after it was built. In particular, one machine could imitate another at good efficiency, a property which was used by IBM to "emulate" earlier machines on its 360 models.

Considerable experience with production of the first generation of transistor machines had also made it possible to modify the previous insistence on worst-case design. With knowledge of actual variances and distributions of component parameters it became feasible to use statistical design rules in which the concatenation of unfavorable tolerances could be made very unlikely. The higher was the degree of integration the more this kind of knowledge could be exploited in specifying the design unit.

SEMICONDUCTOR MEMORY

The advent of semiconductor memory closes a circle. ENIAC had used the same technology, vacuum tubes, for both memory and logic and now semiconductor technology provides the same commonality. In between has been the electrostatic memory and the ferrite core memory, both were important.

Semiconductor memory is now leading LSI. The regularity and simplicity of memory arrays and their interwiring allows the density of memory bits per chip to exceed logic circuits per chip by a considerable margin. In a curious inversion memory, which had been the difficult thing at the outset, has become the easy thing. Even more interesting is that semiconductor memory can be considered a fusion of its two immediate predecessors. The basic idea with electrostatic memories was to use the high resolution of an electron beam to put many bits on a small surface; and the basic idea with ferrite memories was to fabricate a structure for each and every bit but to seek maximum automation of that fabrication. With semiconductor memories the high resolution of the electron beam will be used to fabricate on a small silicon surface the bit-by-bit structure of a very large memory—and with the full automation inherent in the LSI process.

Semiconductor memory will replace ferrite memory on both cost and capacity grounds. It has already provided a less obvious but fundamental change in computer design. The old dream of infinite memory with infinite speed, fostered no doubt by the incredibly limited memory of early systems, gave way to the more realistic appreciation that a combination of a small fast memory with a large slower memory could be automatically managed to act, statistically, as a fast large memory. This change-of-kind in memory organization finally matched the logic change in going from tubes to transistors. The memory-logic speed gap which had prevailed with ferrite memories and had caused so much difficulty with high performance systems is now significantly improved.

REFERENCES

1 R SERRELL et al
The evolution of computing machines and systems
Proceedings IRE Vol 50 No 5 1962

2 N NISENOFF
Hardware for information processing systems: today and in the future
Proceedings IEEE Vol 4 No 12 1966

3 J P ECKERT JR
A survey of digital computer memory systems
Proceedings IRE Vol 41 No 10 1953

4 J A RACHMAN
Computer memories: A survey of the state-of-the-art
Proceedings IRE Vol 49 No 1 1961

5 L V AULETTA et al
Ferrite core planes and arrays: IBM's manufacturing evolution
IEEE Transactions on Magnetics Vol MAG 5 No 4 1969

6 W BUCHHOLZ editor
Planning a computer system: Project stretch
McGraw-Hill 1962

7 C A ALLEN et al
A 2.18 microsecond megabit core storage unit
IRE Transactions on Electronic Computers Vol EC 10 June 1961

Mass storage—Past, present and future*

by ALBERT S. HOAGLAND

IBM
Boulder, Colorado

INTRODUCTION

Mass storage as a functional need in computer systems is continually increasing in importance with the growing trend to interactive terminal-oriented systems, serving to store a systems data base and resident programming systems. The associated capacity, plus the ever expanding magnitude of such information, far exceeds the range where "electronic" memory is economically competitive. Included in the product category defined as mass storage are disk, tape, and card-like recording structures. We see in the future the perspective that at a growing number of installations all data will be available on-line under systems control.

Extending the range of information processing involves new applications that generally call for manipulating large masses of data. Thus, the need for mass storage capabilities with reasonable access time tends to be open ended with the supply—at an economic price—lagging the demand.

I believe that magnetic recording will remain the technological base for mass storage for the foreseeable future and that continued progress commensurate with that realized in the past will be realized during the seventies. This position will be articulated later on in this paper as well as an analysis of alternate technologies that have been proposed.

Fifteen years ago the highest storage density in a commercial disk file was 2,000 bits per square inch, while today (early 1972) this figure is approximately 800,000 bits per square inch—(greater than two orders of magnitude improvement). However, I feel eventually magnetic recording storage densities far in excess of 10^6 bits per square inch will be realized—a point of view reinforced by the progress previously projected and now already achieved.[1]

Further, magnetic recording enjoys a broad worldwide base of technical expertise and associated manufacturing facilities and derives intense stimulation from both computer storage needs and the audio/video recording marketplace.

The success of the disk file has been remarkable and placed this device in such a predominant position that it has essentially become synonymous with direct access storage (See Table I).

In the last decade significant advances have also been made in the performance of conventional (1/2 inch seven and nine track) tape drives as well as in the elegance of their design for reliability and serviceability. However, we are concerned here with the future nature of mass storage subsystems and these design advances do not essentially change the perspective of standard tape drives in an overall sense. The large expansion of tape libraries has already stimulated the development of specialized mechanical hardware for the automated handling of standard tape reels even though this step, not being conceived of in the original design of tape drives, results in unsatisfying implementations. However, we are beginning to see the emergence of flexible media storage devices, departing from the compatibility constraints of 1/2 inch tape, where the emphasis is on low cost per bit on-line—achieved by high areal density. Examples are the TBM of Ampex using rotary heads and the Masstape of Grumman, the latter also introducing the cartridge or cassette concept for accessing reels mechanically under system control.

Overall, technology appears to have advanced as fast as projected while the systems organization and use of mass storage hierarchies still retains much fertile ground for sophisticated design and application.

FUNCTIONAL CONSIDERATIONS

Computers that manipulate data are primarily limited by the number and size of files that can be made readily accessible for processing. Further, mass storage

* Slightly revised version of paper presented at the U.S.A./Japan Computer Conference, October 1972, Tokyo, Japan.

TABLE I—Direct Access Storage—Disk

Year	Model (IBM)	Storage density bits/in^2	Linear density bpi	Track density tpi
1956	350	2,000	100	20
57				
58				
59				
1960	1405	8,000	200	40
1961	1301	25,000	500	50
62				
1963	1311	50,000	1000	50
1964	2311	100,000	1000	100
1965	2314	220,000	2200	100
66				
67				
68				
69				
1970	3330	800,000	4000	200

in 15 years x 400 bits/in^2
x 40 bpi x 10 tpi
(precursor: magnetic drums @ 100 bpi and 10 tpi)

devices are now called on to fulfill important systems functions. Direct access units are used in compiling and assembling programs where their ability to reach large directories and subroutines rapidly is necessary for responsive program-preparation and execution. For time sharing, programs and data of many users must be stored on-line, to be run as requested in accord with some "optimum" allocation of facilities.

Capacity, access time, data transfer rate, and cost per bit are the basic performance characteristics of mass storage devices. Each mass storage unit has its own particular attributes, and many applications require that a hierarchy or mix of devices be connected in the same computer system. The short access times and high data rates of fixed head drums or disks save on main memory size. However, the higher cost per bit of these devices generally makes them too expensive for file storage, and moveable head disk structures of higher capacity (and lower cost per bit) but longer access time are used. Generally, the lower level in a hierarchy includes tape for the archive and data whose processing can be well scheduled in advance since further economies in cost of storage are possible at the expense of even slower access times.

Systems-derived performance factors such as through-put depend not only on the mass storage specifications but also on indexing procedures, memory allocation and record chaining provisions, file activity, provisions for access queuing, error checking techniques, etc. Thus, the associated control logic is an integral facet of any storage subsystem.

TECHNOLOGICAL CONSIDERATIONS

The continuing need for economical high capacity storage has required the intensive exploitation of magnetic recording technology for implementation. Thus, storage units involve the physical integration of recording media, transducers, precision mechanics, servo systems, and electronic encoding and decoding techniques to achieve a meaningful set of capacity/access time tradeoffs.

Access to any data location is provided by relative motion between the storage surface and an associated transducer able to record signals on and sense the state of the storage medium. A single transducer may service many data "tracks". The recording density (bits per square inch) is principally a function of the positional tolerances (spacing, tracking, and clocking) that can be maintained between the storage medium and the coupling transducer. The access time variability to memory locations arising from the requisite motion necessary to scan large areas, makes the data organization of a mass store a key factor to effective systems utilization.

Magnetic recording is predominantly a surface area phenomenom. A magnetic head magnetizes a thin layer of magnetic material traversing the small region adjacent to the head gap in accord with an applied write current and provides an induced voltage, reflecting the rate of change of recorded magnetization when scanning on readback. A track is defined by the head width in the direction normal to the direction of relative motion.

Extremely large capacities will dictate a thin flexible substrate that can be tightly packed (or rolled). Note that for 10^{12} bits of data at 10^6 bits/in^2 we require a million square inches of media. We are interested in capacities greater than this number. Thus, even at much higher storage densities we still must mechanically access thousands of square feet of storage material.

The average random-access time will range from milliseconds to seconds because mechanical motion is required. Multiple access paths (read/write assemblies) is a method to improve the overall access rate. The greater the bit storage density (and the number of bits associated with a given read/write transducer) the lower the cost/bit and, correspondingly, the shorter the average access time for any *given* capacity. Areal density thus can be viewed as the figure of merit for recording storage when similar configuration designs are considered.

Although it is only in recent years that magnetic recording has come into wide general use, its invention by the Danish engineer, Valdemar Poulsen, dates back to 1898. The paramount functional advantage of magnetic recording is the reusability of the recording medium. This property permits the modification or updating of stored information. Additional advantages of magnetic recording for mass storage are: the simplicity of recording transducer (read and write); flexibility in configuration (and hence, choice of performance specifications) due to the ability to place a magnetic layer on almost any supporting surface in conjunction with the ease of mounting a magnetic head or heads; the high bit storage densities and read-write transfer rates obtainable from the magnetic recording process; and relative ruggedness with respect to handling and environmental conditions. The further features of replaceability and library shelf storage of the medium (e.g., tape reels and disk packs) make this mass storage technology extremely attractive and very economical.

Magnetic recording represents the integration of several basic engineering fields and has been generally characterized by rapid progress achieved by evolutionary advances rather than dramatic innovations. The one "breakthrough" that can be identified with the computer field is the "air-floated" head. Otherwise, advances in the magnetic recording art have largely emanated from increasingly higher precision and quality in components and sophistication in recording electronics.

HISTORICAL GROWTH AND PRESENT STATUS

The original work which ushered in mass data storage was firmly under way by 1947. This activity was associated and concurrent with the explosive "take-off" of the digital computer field at that time. Early work was oriented to the needs of scientific computing. The storage device was a magnetic tape drive, to provide both an auxiliary "back-up" storage for main memory and for terminal buffering (data rate "matching" between input/output equipment and the central processor) in large-scale scientific systems. The later emergence of commercial data processing brought with it a wider variety of functional needs and mass storage hardware. Magnetic drum memory development for small and medium speed computers in these early years served to significantly add to the technological base of digital magnetic recording.

Commercial or business data processing, as it was evolving as a main facet of activity in the electronic computer field in the early 1950's, gave a tremendous impetus to mass storage development and had a major impact on its direction. File storage for records maintenance became the central requirement.

Magnetic tape was exploited early for business data processing. In updating a file, for example, the master and transaction tape reels are serially read (information is arranged and maintained in ordered sequence) and an updated master tape is created, on another transport, with the unmodified and updated records being transferred and recorded. This procedure was dictated by the fact that a tape reel could not be selectively rewritten. A tape reel is relatively cheap and therefore its use for low-cost, off-line, archival storage became attractive.

The character of much commerical data processing indicated the need for an entirely different type of file storage. The desirability of storing large volumes of information with any requested record available rapidly gave stimulus to the development of a mass direct access store.

Direct access storage involves addressing by physical location to a single record or a particular block of records (one track), which is then scanned. Any record can be read, written, or modified without affecting any other record. The "set of keys" (record identifiers) of a file will, in general, bear no direct relation to the closed set of machine addresses. Various randomizing techniques (key transformations) are used to convert scattered keys covering an extensive range to a dense and relatively uniform distribution of numbers to obtain automatic addressing capabilities.

The air bearing supported head (using an air cushion to control head-to-surface spacing) was the innovation which, associated with the above concept, brought about this entirely new type of device. An air-bearing head can follow considerable surface fluctuation—up up to 100 times the spacing. Since the readback amplitude wavelength dependence on separation is given by $e^{-2\pi d/\lambda}$ (where d = separation and λ = wavelength), high recording densities would be impractical without such a method for establishing and maintaining a close head-to-surface spacing. By this air-bearing spacing technique, it was possible to develop a high-capacity rotating disk array since a magnetic head could then closely follow the appreciable runout of large disks.

The first version (the RAMAC, announced in 1956) could store five million characters with an access time to any record of less than a second, having one head mechanism servicing the entire disk array. Secondary technical features of note were the use of self-clocking and a wide-erase narrow read-write head unit. These design approaches, combined with the use of the air-supported head, provided techniques to achieve adequate head-to-track registration in such a mechanical structure, permitting the high track density and high bit density necessary to realize a large capacity.

Initially pressurized air was fed into the head-surface interface to maintain separation. Around 1960 a significant advance was achieved as self-lubricating air bearings came into general use on both disks and drums, bringing great simplicity and cost advantages.

Many approaches to card-like direct access mass storage devices have been undertaken over the past decade. However, the engineering simplicity of rotational motion coupled with linear actuators has favored disks.

The next major innovation in mass storage (1962-63) was the replaceable disk pack concept, which emerged as a practical alternative due to the rapid progress achieved in increasing storage density—from 2,000 bits per square inch (1956) to 50,000 bits per square inch (1962), permitting adequate capacity to be stored on a few small disks.

Discernible themes that have emerged in the field of mass storage:

1. The disk pack file is clearly the principal mass storage structure characterizing modern generation computer systems. However, the use of disk packs as a method of loading and unloading user jobs can be expected to diminish since increasingly systems will view peripheral hardware as a resource which is allocated and scheduled without operator intervention. Further, for efficient system usage a single users data sets are frequently spread among several packs (or volumes) interspersed with data associated with other user jobs.
2. Multi-access assemblies. By overlapping accesses, the mean access time can go down with increasing capacity at a relatively fixed cost per bit. The performance region of several hundred million bytes of data with access in milliseconds is a key to many on-line "data-bank" type information systems and the appeal of such applications has spurred the rapid trend to such file "facilities".

The "state of the art" magnetic recording density for production disk files has now moved to the region of 10^6 bits per square inch, a factor of 500 greater than the storage density of the first commercial disk file. Major further advances in density and thereby in lower cost per bit are clearly indicated based on reported laboratory investigations. Techniques that could make equivalent strides in reducing "hardware" access time are unrecognized.

Among specialized mass storage systems recently developed are the Photo-Digital Cypress (IBM); a trillion bit storage system for the Livermore AEC Laboratory based on electron beam recording on photographic film—with a flying spot scanner for reading; and Unicon, a recording scheme which stores information in a plastic film with a laser by "drilling" holes (whose presence or absence can be optically sensed). These beam addressable storage systems do not allow update in place. Their introduction in spite of this limitation reflects the low cost per bit (for immense capacities) from a high areal density—derived from simplicities in implementing high track density inherent in beam recording. However, continuing advances in areal density have led magnetic recording technology to move into this domain as evidenced by the TBM, an Ampex trillion bit store based on video magnetic recording techniques exploiting a rotating head configuration.

FUTURE TRENDS

Each user wants all data immediately available to him. The appetite for more storage capacity with high speed access appears as insatiable as the demand has been for more computer power.

While a storage hierarchy arises as a compromise between accessibility of data and cost of storage, the user in the future will not want to even be aware of its existence since his focus is on problem solving.

Further, shared data and security provisions must be systems managed. Thus, much sophisticated systems development work is still necessary if we are to fully exploit our growing technological capability.

A major storage subsystem problem is determining the proper balance between storage devices. We need to better understand the flow of data and develop design guidelines, recognizing that the access "gap" from electronic memory to electromechanical storage devices will cover a relative access speed "differential" of 10^3 to 10^5. Storage control logic is now beginning to extend to the functional ability to automatically stage records within the hierarchy of memory/storage. Much empirical data is now being analyzed and simulation studies carried out on such measures as "miss" ratios (i.e., the relative probability that a given request by the CPU may not be found in the first level of a hierarchy, etc.) to better understand the way a hierarchy influences performance and affects data organization.

A major advance in hardware reliability is urgently needed if the full potential of mass storage devices is to be realized. Capacity-access time improvements cannot be at the expense of reliability because we are now asking users to place and maintain all their vital records on-line under computer control.

The storage hierarchy of the future now appears to include a relatively vast semi-conductor memory with an access time of a fraction of a microsecond; disk files at the next level with a much larger capacity and the order of 10 milliseconds access time; and finally a low cost on-line tape-like (flexible media) library; the latter also serving an archival role and whose capacity will essentially be the capacity of the storage subsystem.

Advances in semi-conductor or magnetic bubble technology will displace fixed head files in time since this cost/performance range (as well as that of low capacity mechanical storage devices) is exposed. The capacity of a recording type store must allow the cost of the basic mechanics to be apportioned over a bit volume sizable enough to yield a cost per bit significantly below those achievable by electronic memory if the technology is to be viable.

ALTERNATE STORAGE TECHNOLOGIES

No technology can be improved by significant stages indefinitely—eventually some limits will be encountered where the investment to make further advances exceeds the return that can be realized. However, technological potential is assessed by a fundamental understanding of the physics of storage phenomena and the associated design parameters and *not* by whether it's "new" or "old." Here we find that many alternate technologies are discussed and projected on the basis of their ultimate capabilities (in terms of optical wave-length, etc.) while digital magnetic recording tends to be identified performance-wise only by what we have achieved. However, theoretical and experimental studies of magnetic recording show the potential for major further advances in areal density. Work on alternate technologies must better recognize this "moving target" situation than has been evident from the literature if the resource expenditures are to offer a realistic hope of pay off.

An alternate technology can be viewed strictly as a replacement or by its unique characteristics also invoke changes that will impact systems architecture. Any new technology in effect must bear an entrance fee to set in place its technological base and possibly in addition a systems base essential for its exploitation.

In spite of dramatic cost reductions projected for LSI semi-conductor arrays and magnetic bubbles, their costs will still be significantly above recording storage when capacities reach the range of 10^8 bytes so that they cannot be considered competitive for the mass storage area. The sharing by mechanical motion of the transducers and electronics over many bits yields the low costs in recording storage, recognizing the access time penalty implied. With semi-conductor approaches one must provide electrical connections to each bit location. Magnetic bubbles save in this regard but still require the fabrication of a pattern at each cell location for its definition and access interaction. (Magnetic bubbles exploit a shift register type of organization. Thus, access time involves on the average one half the recirculation time of a storage loop, which can be hundreds of microseconds.)

Some beam addressable recording techniques have the unique potential for microsecond access to densely recorded digital information within the field of view of the beam, defined by electronic deflection methods. However, the field of view tends to be inversely related to bit resolution and therefore for mass storage one cannot avoid media handling. Thus beam addressable technologies still require mechanical accessing (or access times from milliseconds on up). Areal densities are comparable to those projected for magnetic recording and no really suitable medium has been developed. The medium appears the limiting factor in recording-type storage and magnetic recording offers the greatest flexibility in material selection.

Electron beam spot sizes less than one micron in diameter have permitted writing (in a vacuum) at a density of several million bits per square inch (approximately 2000 bits/in. by 2000 tracks/in.) on silver film. Read-back must be done optically at a separate station after film processing. A significant capacity/cost advantage must be achieved in such a read-only type store to justify the acceptance of the systems performance limitations on updating, posting, and immediate write-checking.

Optical techniques offer possibilities for simple parallel data flow to a block of information as well as the prospect of "distributed" bit storage (e.g., by holography) where the effects of media defects and registration misalignment at high densities may be minimized. A hologram is created from the bit pattern and recorded. The original image is reconstructed for read out by photodetectors.

In magneto-optic schemes writing is done by application of a focused laser beam to the selected spot with the simultaneous application of a magnetic field. The laser beam heats the surface locally, reducing the coercive force. The magnetic field then is able to switch the magnetization. Subsequent cooling raises the coercive force and renders this change stable. Reading is based on sensing the Faraday rotation of the polarization angle of a laser beam. The optical rotation is a function of the direction of magnetization. Since the constant magnetic field covers a "large" region, to alter the state of magnetization the data must first be reset to "0" and then on another scan the "1" bits can be selectively inserted.

Certain unique applications have led to the development and use of a very limited number of specialized mass store systems exploiting an alternate technology. While it is sometimes possible to tailor a particular technology toward a specialized application this provides a narrow cost/performance range of applicability.

Thus, it is very unlikely beam addressable recording will displace conventional digital magnetic recording technology and is itself threatened as the prime contender in the search for an improved storage technology by the rapid advances in semi-conductor and magnetic bubble memory work.

For the storage and retrieval of strictly human readable information there will continue to be developments of microfilm as well as in techniques to digitize and compress such images for both storage and the eventual attractiveness of possible automatic processing applications.

MAGNETIC RECORDING TECHNOLOGY

In measuring a storage technology, areal density is the key figure of merit to consider. Lower cost/bit is a direct consequence of storage density increases. While hardware "overhead" limits the ability to extend mechanical storage configurations into small capacity versions—an area threatened by large scale integration with semi-conductor memory or possibly magnetic bubbles—continuing cost per bit reductions are possible with increasing capacity.

Magnetic head design skills have advanced to a point where the principal concerns are to evolve batch fabrication techniques that will provide precision assemblies on a continually reduced dimensional scale. This has stimulated thin film head technology which is emerging as a promising new avenue for head array design and fabrication. This approach lends itself to precision batch fabrication through photolithography and materials process technology akin to the methods pioneered by the semi-conductor industry with the consequent possibilities for cost reductions.

Magnetic heads now have the capability to resolve in excess of 50,000 cycles per inch with frequency bandwidths extending considerably beyond ten megacycles. Head gaps are now approaching 10 microinches as manufacturing techniques improve (the first head gap null moving out to approximately 100,000 cycles per inch) and the introduction of thin film structures will greatly extend the usable bandwidths. It does not appear that the head will be the limiting factor in performance but rather the storage medium.

The mainstream effort in high density magnetic surface work is to achieve higher quality (uniformity, smoothness, etc.) recording surfaces. To date, the most common medium is a particle dispersion in a binder so that magnetic and mechanical properties can be tailored. (Magnetic films in use today are oxide coatings formed from a dispersion of either Fe_30_4 or γ-Fe_20_3 in an organic binder, and Co-Ni plating). Media magnetics are secondary compared to mechanical properties and quality. Higher areal densities require lower noise and smaller particles in improved dispersions or new approaches to a "continuous" film medium must be pursued.

Thinner magnetic recording films are indispensable for higher density whenever pre-erasure is not feasible. High bit density is synonymous with close spacings and smooth surfaces are essential, both for air film lubrication and to minimize the generation of any wear debris. Medium defects are a further source of noise that must be reduced, recognizing that at 10^6 bits/in.^{2} a bit is allotted only a millionth of a square inch.

To date, linear density has been increased by scaling down three geometrical parameters; head gap, head-to-surface spacing, and medium thickness (where pre-erasure is not possible), and this theme will continue to be the name of the game. The "swinger" is the spacing.

Reduced head-to-surface spacings will bring about a further large increase in linear bit density. The resolution of flux transitions on the surface scales with head-to-surface spacing; the head gap and recorded depth of penetration being proportionally reduced. 50,000 bpi now appears within our grasp at spacings of 5 to 10 microinches. (The mean free path of air is approximately 3 μ in at atmospheric pressure and with microinch surface finishes this spacing capability appears feasible with acceptable bearing and/or wear characteristics.)

Track seeking and following servo access techniques, which can circumvent the track density limitations imposed by the build-up of cascaded mechanical tolerances, will be applied to effect significant increases in track density. We can project an adequate signal to noise ratio with much narrower tracks than we can live with today. Thus, mechanics still offer considerable room for further advances in storage density. To continually improve track registration tolerances will require servo techniques that depend upon interspersing or superimposing data and servo control information and more sophisticated head structures.

The calculated density limit based on a typical pulse response with standard recording techniques under ideal conditions is extremely high. However, (even theoretically) a large reduction in this figure results from a simple consideration of variability between magnetic heads, tolerance on magnetic coating thickness, variations in spacing and speed, head-to-track registration stability, etc. Track mis-registration tolerances require the introduction of a guardband which reduces the available read signal. In addition we must deal with "noise" sources, such as surface imperfections, etc.

Recording methodology is as yet an inadequately explored means to further upgrade storage density. Since the advent of digital magnetic recording for mass data storage, the primary avenue taken to increase density and thus performance has been through improved recording mechanics. This work emphasizes improving the "resolution" of a recording system.

The application of advanced communications techniques to the magnetic recording "channel" may yield significant gains in bit density relative to pulse resolution. Much more sophisticated error detection and correction codes are indicated (the choice of such codes also involves timing tolerances on clocking) with the move to higher and higher areal densities.

Signal-to-noise calculations indicate much higher storage densities are realizable. Eventually density limits should be bounded by intrinsic media and amplifier noise. Increasing data rate per channel (or relative velocity) is equivalent to increasing bandwidth and since electronics noise increases with bandwidth this design direction will place some limits on density.

During the seventies we see a potential advance in areal density of almost two orders of magnitude.

SUMMARY

At the present time there is no approach to mass data storage that could obviate the use of physical motion, using continuous recording media and transducers. While a completely electronic mass data store is an obvious goal, such memories seem possible only on the low capacity, high-speed side of mass storage (now filled by drums and disks with fixed heads). The storage hierarchy of the future will include disk and high areal density flexible media with all storage on-line and under systems control in a growing number of installations. There are substantial reasons to believe magnetic recording will undergo another decade of progress comparable to the past one, although there is something in man that rebels against the necessity for mechanics in purely *information* storage and processing systems, our attitude apparently formed early from bitter experiences of breakdown and poor reliability (peripheral-wise, automobile-wise, etc.).

For the foreseeable future, however, it is clear that mass storage based on digital magnetic recording has a vital and increasingly important role in information processing. Further, we see the potential of increases in storage density of two orders of magnitude during the next decade.

GENERAL REFERENCES

1 A S HOAGLAND
Mass storage revisited
Proc FJCC 1967

2 J M HARKER HSU CHANG
Magnetic disks for bulk storage: Past and future
Proc SJCC 1972

3 W A GROSS
Ultra-large storage systems using flexible media, past, present and future
Proc SJCC 1972

4 T H BONN
Mass storage: A broad review
Proc IEEE 54 1861 1966

5 L C HOBBS
Review and survey of mass memories
Proc FJCC Spartan Books Washington D C p 195 1963

6 R E MATICK
Review of current proposed technologies for mass storage systems
Proc IEEE 60 266 1972

Software—Historical perspectives and current trends

by WALTER F. BAUER and ARTHUR M. ROSENBERG

Informatics Inc.
Canoga Park, California

SUMMARY

The importance of the study of history is to understand the forces which produce the events. Thus possibilities and prospects for the future may be better understood and evaluated. The development of historical perspectives as opposed to a historical record seems to have advantages in achieving that goal. This is especially true of programming where the award of historical "firsts" seems to be difficult at best.

This paper traces the historical evolution of software in the context of the developing computer hardware technology. It has been roughly only twenty years in which concepts of software have evolved from very crude beginnings into a sophisticated element of the expanding computer industry. It is time to take stock of the past and start to focus on the practical directions that software, as a technology and as an industry, will take.

The genealogy of software is developed, starting from the early 1950's when the most elemental concepts were subroutines, simple assemblers and simple diagnostic programs. From these simple concepts, through a process of successive additions of complex structures, came compilers, complex operating systems, and on-line systems programming. A newer concept, the "software product," is discussed including its business implications. A related subject, the development of machine aids to programming, is also a topic of discussion.

Advanced compilers and software language apparently come from the lineage of subroutines and simple assemblers. Advanced operating systems seem to have more in common with early interpretive programs than with any other possible antecedent. It seems clear that modern time-sharing systems were affected greatly by early on-line systems accomplished as part of the first military systems and, before that, from early diagnostic programs for finding programming and machine errors.

The future will probably not see any more proliferation of universal languages. For example, in the rapidly growing business data processing world, COBOL will continue to be used with COBOL shorthand and COBOL generators increasing in importance. A language-compatible set of tools for debugging, maintenance and documentation will be emphasized. Applications-oriented tools will receive much more attention. The trend starting with early report generator techniques and continuing with the powerful file management system approach will flourish. Machine emulation and software transferability will become of paramount importance with the relative increase of software over hardware costs.

Although important progress has been made, it is a general indictment of the programming profession working in industry that there is such a great time gap between concept and practical reality of a new idea.

INTRODUCTION

Computer programming is a unique and modern professional activity. The end product of programming is software. Programming is the process of developing procedures for a computer. The procedure itself is a piece of software.*

The computer is unique. It is an instrument of great versatility and universality. In the history of technology it is difficult, even impossible, to cite a preceding development so universally applied. Programming, which makes the computer useful, is equally unique. The computer, programmed appropriately, is a universal and powerful device for handling information.

There is little in man's history that can be cited as a direct antecedent of programming. Computational pro-

* The term, software, is sometimes taken to mean all activities, processes, and procedures surrounding the modern computer; in this definition, it includes everything related to the computer except hardware.

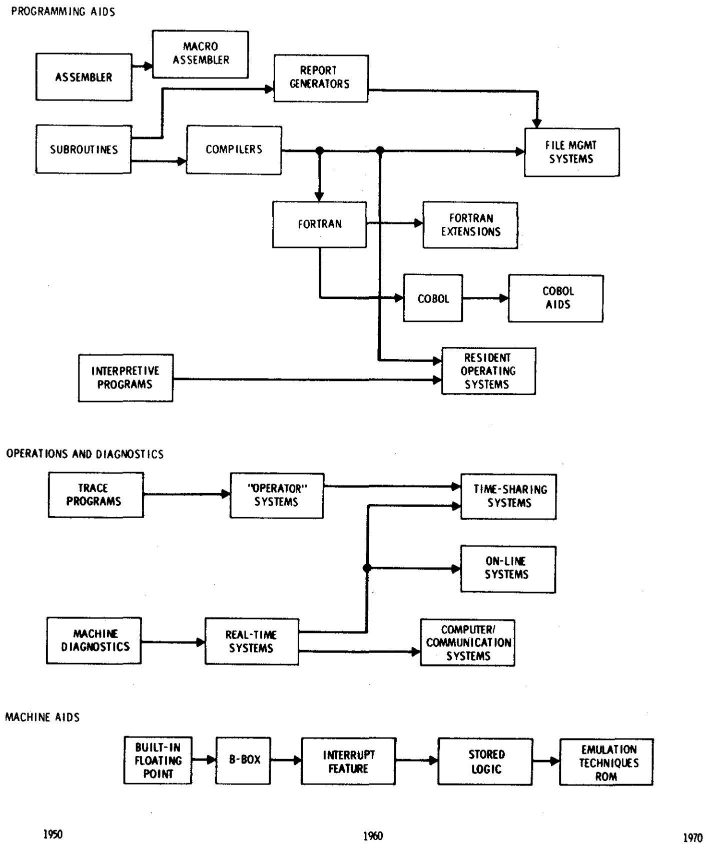


Figure 1—Early system geneology

cedures of all kinds, including those for desk calculators, offer some precedent, as do office procedures for business data. Further precedent is found in formalized rules for musical composition, first stated in modern terms by Bach in "The Well-Tempered Klavier," and in the setup procedures for such machines as the Jacquard loom. Precedent may also be sought in the rules of certain games which require playing by a procedure to win, such as tic-tac-toe, mill, and o-wah-ree. Such games extend back into human prehistory.

A good deal of the power and versatility of the stored program electronic computer is found in its ability to alter its future procedures on the basis of results of past procedures. This ability of the computer gives it that automatic character which permits it to carry out exceedingly involved and complex tasks, guided by interrelated program structures.[1]

The writing of a machine language program for the computer of typical design is a straightforward, but monumentally detailed and onerous task, especially if done with no aids. Programmers of the earliest electronic computers soon tired of such work and chafed under its limitations. They were led, quite naturally, to use the computer itself to make programming easier and more versatile. The development of programs on behalf of programming itself—to make programming less burdensome and expensive—got under way. These activities were called "automatic programming" in the middle of the 1950's.[2]

In the early 1950's, the process was referred to as the preparation of utility and service programs; the "automatic programming" label came later. Nowadays the practice is usually referred to as "systems programming." Whatever the name, the process is one of developing languages and procedures, and computer programs to operate in conjunction with these languages and procedures so that the computer itself can be the principal instrument to make programming easier.

EARLIEST CONCEPTS AND OBJECTIVES

The overall objective of systems programming is twofold: to make programming easier and to make the machine run more efficiently. Most of the programming sophistication of the early 1950's was aimed at programming efficiencies, although the great shortage of machine time, and its great cost, put heavy pressures on systems which used large amounts of machine time.*

Four areas of the programming task were targets for development of programming efficiencies. These were: the conceptual phase, consisting of architecture, design and specification; the programming or coding phase; the checkout phase; and finally, the modification and program maintenance phase. The earliest system software was aimed at making the coding job more efficient. A simple example is found in the replacement of the machine language operation code with an operation mnemonic.

In later years the targets for improved efficiencies turned to other ares such as design, program checkout and program modification. Thus, the Holy Grail of the programmer has been the development of systems to make programming efficient; but, the statement of the problem is more complex than that. For a given set of data processing functions and for a given degree of com-

* The question of objectives of system programming is an interesting philosophical point. In modern systems is the objective to make the computer run faster and keep programming tractable? Or is the objective the other way around—make programming easier and keep machine efficiencies reasonable? It would appear that a strong case can be made for the following: The 1960's, with increasingly complex applications and machines, systems programmers, mostly as a defensive measure, emphasized machine efficiencies to insure that programming became more efficient in complex environments. That notwithstanding, the authors believe unqualifiedly that in view of high labor costs and decreasing machine costs, the objective clearly should be the increasing of programming efficiencies.

plexity of the data processing problem, the programmer wishes to make his programming easier and to make the machine run faster. Success is elusive since new solutions must be found in an ever-increasingly complex environment and an ever-increasing set of data processing functions to be performed.*

SOFTWARE GENEALOGY

A historical perspective of how programming developed may give us better understanding of present and future developments in the list of the past. While citations of early milestones are given, no attempt is made here to record history per se, as others have done.[3] The concern is with the major events of programming development over the years, not with attributing invention or giving credit for milestones achieved. Typically, no matter what the programming technique and the alleged development time, it is possible to find someone who claims the same accomplishment at an earlier date.

There is often little unanimity of credit for earliest development. The time of conception of an idea is uncertain, and it is at least debatable whether the milestone is achieved upon conception or upon execution. To a considerable extent, programming structures are unclear, relations among programming methods cloudy, and programming elements themselves not always characterizable. Notwithstanding these difficulties of being able to distinguish, it is instructive to consider the main directions of programming development. One looks for the essence of what was developed, why it was developed, and, ultimately, what the future may bring.

The earliest of system programming activities seem to be divided among subroutines, interpretive routines, assemblers and loaders. To these may be added early attempts at diagnostics, both hardware and software. Machine aids to programming, that is, hardware design changes to make programming easier, have also come with some frequency beginning early in computer history. While this latter topic is not, strictly speaking, a programming development, it is related enough to be considered. It is as true today as it was twenty years ago: programming and machine design and characteristics are inextricably related; even today's "applications" programmer must know hardware characteristics. Thus, the three main channels of development can be characterized as the search for language power, the search for diagnostic power, and the search for machine aids to programming.

Figure 1 is a chart showing the genealogy of major software developments, emphasizing the earliest periods and how the earliest developments gave rise to later ones. While there is no claim made here for comprehensiveness or precision, the figure can serve in the purpose of discussing the structure and evolution of programming.

The beginnings of the development of programming structures and systems can be traced back to the emphasis of subroutines—their structure and use. The subroutine idea came about, naturally and quite obviously, by the programmer's observance that certain segments of the code were used or could be used repeatedly. From then on, ideas grew in proliferation. Macro instructions and pseudo instructions were developed as a convenience for calling subroutines and to specify the parameters to be used by the subroutine. One of the earliest technical writings on the subject of subroutine use and structures was that of Wilkes, Wheeler and Gill.[4] Needless to say, the development of the subroutine idea gave rise to the compilation of many subroutines into a running program and therefore to a major enhancement of simple assembly language programming, as well as to the compiler concept.

There seems little doubt that early programmers were led to interpretive programming as a simple extension of the subroutine idea. If, instead of occasional transfer of control to a subroutine and the interpretation of a macro or pseudo instruction, one considered that the entire running program consisted of macro instructions which are to be "interpreted" by the subroutines, a wide range of possibilities and a language of considerable power could be developed. Among the first interpretive programs were the ones for floating point arithmetic.

It is interesting to digress for a moment to discuss the similarities between the earliest subroutine structures and today's complex operating systems. In the earliest square root subroutine, for example, one parameter was supplied; namely, the number from which the square root was to be extracted. Today's operating systems are complex structures consisting of many subroutines. During the course of using the operating system, many parameters are supplied to the operating system for correct action: the programmer supplies the file from which information is to be extracted; he supplies the part of the program to be activated in the case of an interrupt on a certain channel; and, he supplies the output device as well as the form of the output desired.

This then characterizes programming and the development of programming systems: simple structures are used in multiplicity to create complex structures; in the building of these structures there is a layering of simple functions, each interfacing with others and using

* "The problems of programming at this point appear as yet mostly unsolved. . . ." Report of the Discussion Group on Universal Codes. Proceedings of the First Joint AIEE-IRE Computer Conference, Philadelphia, 1951.

the capabilities of the layer previously supplied. Interestingly, the overall structure has proceeded to the point where the typical applications programmer knows little about the internals of the operating system he uses. To anyone trained in science or technology, this comes as no surprise; all of science and technology is the building of complex structures and theories from earlier simpler theories and structures. This is the Scientific Method.

Beyond the subroutine idea, the idea of the compiler quite naturally evolved. Since a larger number of subroutines were in existence, instead of the compilation of these subroutines manually into the program prior to running, it seemed natural to make the machine assemble all subroutines needed and at the same time create all necessary connective tissue to insure a coordinated, concerted data processing effort. Emphasis then shifted to the language—its use and its power. Attention also shifted to the development of assembly programs or compilers which would preprocess all programming information and create the running program. The programmer saw that, through the use of the subroutine-assembly program idea, he could write in decimal and let the machine carry out decimal-to-binary conversion. The programmer could write in mnemonic instruction codes and let the assembly program translate those mnemonic codes into numerical instruction codes. He saw that he could use symbolic addresses to make programming easier and have the machine later translate them to machine addresses by means of the assembly program. Many of these ideas were developed and explained by Hopper.[5]

Thus, the so-called "one pass" assembly program was born. This program performed the simple functions described above and used "regional coding" so that all machine codes would be assigned during the first pass through the translation process. There was no need for further recursive processing of the data. Incredibly simple by today's standards, these one-pass assemblers represented proud achievements by programmers in the early 1950's.

For a time in the development of system programming there was competition between interpretive methods and assembly/compiling methods. Assemblers and compilers took a source statement program and produced object code, properly location-keyed and ready to run. With a well-done assembly or compiler system, the object code could be efficient. The interpretive approach, on the other hand, used, in essence, "a different machine"—one built of programmed subroutines rather than machine language itself. Interpreters could be very rich in language power, but they tended to be profligate of machine time.

Today the place of the interpreter, except for special purpose interpreters, has been largely taken by the hardware/software design approaches of stored logic, read-only fast memories and microprogramming: "emulation" is the modern word used.

Early interpretive programming put heavy emphasis on applications in instruction and education in programming itself. Interpreters were used to structure simplified machines which could be learned easily and rapidly by the relatively uninitiated.[6] This design approach is seen in still viable form in many of the timesharing system languages of today. Among the early interpreters applied extensively was "Speed Code," an IBM project started in 1953 and featuring operations in floating point arithmetic.[7]

Throughout the 1950's and 1960's, the cost of machine time was always an important consideration. Thus, interpreters tended to remain unpopular. But near the end of the 1960 era, we began to see a rapid rise in the cost effectiveness of the computer. Today the interpretive approach enjoys a resurgence. Today's operating systems provide, through software, or a combination of software and hardware organization, a "new kind of machine," than previously seen and used by the applications programmer. The operating systems of today are closer to the interpretive approach than to the assembler/compiler approach.

Quite another facet of software came from the desire to have the machine assist in hardware and software diagnosis. One of the first of the systems stemming from the search for diagnostic power was the trace program. It was an interpretive program which printed the instruction itself and the results of the instruction if the instruction handled data. Since trace programs often required a long time in execution, selective trace programs were developed which printed only the data that the diagnosing programmer probably wanted to know.

One of the earliest of hardware diagnostics dealt with the problems of the memories of that day. Electrostatic tube memories suffered from having a particular storage location on the tube face affected by actions in surrounding locations. A spot on the tube face could spill over and affect neighboring spots—the "secondary emission phenomenon." Hence came the "read-around-ratio" testing programs. Such a program read around a given spot to determine the frequency of action at which failure would probably occur.

Other kinds of memory diagnostics took sums repeatedly over large segments of memory, assuring, hopefully, that the contents had not altered between successive summings. Similar to the read-around tests for tube memories are today's "delta noise" diagnostics for core memories.

Diagnostics led to an increasing realization that computers were strongly dependent on input/output devices. Early computer programs tended to be long on computation and short on input and output. Diagnostics put emphasis on input/output capability, and these devices too frequently proved to be the ones that needed diagnosing.

Pattern watching on computers with tube memories doubtless gave strong impetus to using a computer to control a cathode ray tube display tube. In about 1952, the famous "bouncing ball" output program was developed for display on output scopes (CRT) of the Whirlwind computer.[8] In this early work we undoubtedly see the first beginnings of on-line systems. For the first time there was immediate and visual reference to what the machine was doing. From this it was a simple step to let the programmer affect the operation of the program, based on what he sees, through the use of sense switches first (the IBM 704), later the keyboards of CRT terminals.

Machine aids to programming, or the design of machines easier to use, is not, in the strictest sense, part of the *res gesti* of the development of today's programming. Yet hardware and system software are so inseparably tied together today that we must consider the internal logic of the computer in a coherent discussion of software.

Among the earliest of additions to machine design was built-in floating point capability. We find, however, that such early, one-of-a-kind machines as the SEAC were equipped with an automatic trace mode.[9] The index register is a common design feature of computers today. It made its apparent first commercial appearance named as the "B box" on an early ElectroData Machine.[10] As old timers know, ElectroData machines later metamorphosed in Burroughs machines.

In the late 1950's, interrupt capability began to appear in computers. This was a most significant development. For the first time, the computer could be "impedance matched" with interconnecting noncomputer equipment. External devices could operate asynchronously from the main frame, and yet all essential communication was maintained.

Imaginative programmers soon developed many uses for the interrupt capability. What began as a narrow adjunct to early real-time systems was rapidly extended to general computer system operation; the interrupt was beginning to be used extensively to interface with common computer peripherals.

The programmer's propensity to interpret and emulate one computer on another through the years led to still another programming aid being built into the machine. Some machine designers reasoned that certain computers would operate more efficiently on certain types of problems if the instruction logic were chosen to correspond to that type of problem. Also they reasoned that emulation would be accomplished better if one could build up instructions from subinstructions—in other words, if there were access to subinstructions through programming. Thus, the idea of microprogramming was invented.[11] In its earliest form, it was not of great commercial significance nor did it receive immediate widespread use. Nevertheless, the important concept of storing in the memory the logic of the machine in such a way that the logic could be changed had a certain appeal and promise for the future.

The stored logic concept was brought into commercial use with the IBM 360 equipments which allowed emulation through microprogramming of second generation (pre-IBM 360) computers. Logical flexibility of the machine and ability to change its programming character, a kind of "subconscious" or "subliminal" level of the machine, has received the official stamp of approval. The idea was to be developed further in the 1960's through the use of privileged instructions through which only certain parts of the memory can be modified and, of course, the important step of the protection of the stored logic through read-only memories.[12]

Throughout the late 1950's and into the early 1960's, there was gradual increase in the recognition that the computer was neither a "scientific calculator" nor a "business data processor," but was, in fact, an information processing device. Scientific problems involving the solution of equations were a comparatively limited activity. Furthermore, scientific applications were usually imbedded in large design problems and engineering applications, problems which required general information handling capability. The recognition of this true character of the computer came about slowly indeed, and likewise the development of programming to reflect this true character of the machine. Programming systems for business data processing applications had been developed on a limited basis by Univac and by IBM in the 1950's, but the breakthrough in programming for business data processing applications did not come until the turn of the decade: in 1960 the COBOL language was specified and compilers were being developed to process the language.[13] An important new era had begun.[14]

PROGRAMMING LANGUAGES

In previous sections primitive programming languages, such as interpreters, assemblers and compilers were discussed. Later developments include programming languages which most directly affect the end users; i.e., those languages used for development of applications. This category includes computational and information retrieval ability on an *ad hoc* basis, as opposed to a predefined application program. Although *ad hoc* "programming" for one-time needs has certain similarities with the general category of program development, it is important to recognize that there are differences in terms of the user of the language, the facilities required, and the mode of usage (e.g., batch vs conversational, compiled vs interpretive, etc.). Basically, the "one-time" program should require less development investment than the continually productive program because of the short-term payoff involved. Thus, we trade off simpler programming language against efficiency of the object code produced. There are no prizes for *coding*; the results are what count.

Assemblers reached a language level approaching that of compilers with the facility for employing existing routines or macros. The macro assembler, in effect, permitted creation and usage of a higher-level language than that of symbolic assembly language. Nevertheless, assemblers produced object program code (instructions) which could modify itself and could be recognizable as source code by the originating programmer. Compilers, on the other hand, generated assembly language or object program code which was not easily recognizable to the source language programmer.

A principal goal of the "higher level" languages was to allow "machine independence" or, at least, to permit ease of transferability from one computer to another. The high point of this interest was reflected in the academic exercise with UNCOL (Universal Computer Oriented Language), as a means for machine-to-machine translation.[15]

There were many "languages" by the end of the 1950's, but only a few generally accepted by the user community for practical, operational use.[16] Needless to say, this acceptance is based on practical, *de facto* considerations, such as government or IBM endorsement and support. The acceptance of FORTRAN by the programming community was very significant because it helped reinforce subroutine library concepts, calling sequence standardization, etc. FORTRAN was designed primarily for scientific applications of numerical nature. The language was procedural with an emphasis toward algebraic notation and functions and the handling of matrices. FORTRAN did not accommodate parts of machine words or character manipulation.

ALGOL was a more definitive attempt at a computational language that was consistent with mathematically "clean" notation.[17] The ALGOL specifications generated a variety of subset language implementations, some of which still exist and are operating today. Although very useful for numerical computation, it lacked features required by business data processing, such as input/output facilities and report formatting.

COBOL was designed in order to satisfy the needs of the business data processing community, including an "easy-to-learn" language that was self-documenting, and the separation of data descriptions from the processing and operating environment.[18] PL/1[19] was aimed at a most comprehensive spectrum of programming applicability, scientific, business and system programming (including "real-time" interrupt processing). Because of its ambitious scope, it has taken awhile to gain its current operational acceptance. In the time-sharing environment, easy-to-use languages were developed, notably JOSS,[20] BASIC,[21] and APL.[22] JOSS was an interactive language system for small numerical problems. BASIC was designed as a simple but effective computational language for use by nonprofessional programmers in the conversational mode; its most notable facility was built-in text editing for creating and changing source code (which JOSS also had).

List processing languages, such as IPL and LISP, have been primarily used in artificial intelligence experimental and research work. The payoff is still to come in future machine architectures.

Each of these languages was designed to function with a compatible operating system but not with another language per se. (Some compiler implementations allowed for in-line assembly language code.) Thus, each language as a "system" had the following set of functional needs to account for, in addition to computational code generation and execution control needs:

1. data definition, file description—In order to properly handle various types of data and their logical storage organizations.
2. debugging facilities—Necessary for any application development effort at the *source language level.*
3. documentation—Required for proper maintenance of production programs and as a debugging tool.
4. output format control (reports)—This is an area of "programming" which is functionally different from computational programming.
5. Input/Output interface—Although there is supposed to be a maximum facility for "device independence," the realities of various input/output storage and terminal formats (e.g., typewriter or printer vs CRT) require proper handling.

In addition, a language system should offer different data entry, error reporting, recovery and debugging facilities for the conversational mode of operation vs the batch execution.

Every language system design varied in the manner and degree to which it offered these facilities and, in fact, every implementation had differences. What this proved was that programming language design was being accomplished at a time when the need was great but neither the operational environment nor the experience was stable and adequate enough to do a comprehensive job. Much mention has been made about the convergence of scientific and business computing requirements. However, it may have gone unnoticed that on-line (interactive) computing also pushed scientific computation to be concerned with things like character string processing, report formatting, etc. The established programming languages could not really "grow up" until the real world operational environment became more stable.

To cope with language deficiencies that appeared in order to program for new application areas, the idea of "extensibility" was introduced. This was really a takeoff on the macro concept and used in some compilers (e.g., ALGOL, PL/1). The extensible language approach, however, offered opportunity for language proliferation rather than standardization.

Although the above-mentioned languages were "higher level" for expressing computational requirements and easier than assembly language for program development, they did not adequately attack the complexities inherent with data access and storage, and another layer of simplicity was added in the area of secondary data storage or data file maintenance and handling. As the proliferation of machine files increased, both in numbers, usage and structure, a major gap appeared between the operating system, that was primarily concerned with physical data storage resources, and the programming languages that dealt with manipulation of data that the object program could get its hands on. It was up to the programmer to integrate all the protocol for getting the data in and out of files, along with appropriate file structure design and available access methodology. It was also necessary to be very *efficient*, because heavy input/output activity is the primary source of long and expensive machine runs. This kind of responsibility was taken over, in part, by the powerful *automatic* and *default* facilities of "file management" systems, which came into prominence about 1967.[23]

With the powerful automatic and default facilities, the development of *writing of* applications "programs" became much more simplified because most of the coding chores for doing things like validity checking, data file opening and closing, line folding report dating, column spacing, etc., were eliminated. It was not merely a case of providing facilities the programmer had to explicitly use; the use of structured forms and automatic functions meant that the applications developer had only to concentrate on the specifics of the application and was not forced to be a "systems" programmer. In effect, the automatic facilities of "file management" systems provides the kind of capability that was sought after in one of the historical buzz words, i.e., "implicit programming."

The descriptive name "file management" has been abused and confused because of the wide range of functions to which it has been applied. If any part of the data file accessing function were performed by a software package, it was called a "file management" or "data base" system. In reality, there can be complete languages which include all functional capabilities of, say, COBOL, along with file handling and sophisticated report generation facilities on the one hand, and on the other, file organizing and accessing supplements to application processing programs written in a standard programming "host" language. It is interesting to contemplate how comfortably the combinations will fit and evolve in the future.

OPERATING SYSTEMS

Concurrent with the goal of making programming easier for the professional programmer, was the pressure to make computer operations more efficient and reliable. These two major objectives gave rise to the modern day operating system.

Operating systems, monitors, control programs, or executives, etc., became more necessary as computer hardware became more complex; the requirements for sophisticated operating system functions depended largely upon evolving machine architectural facilities. The operating system, as a software facade for the hardware, had to exploit the strengths of machine design as well as make up for deficiencies.

The concept of operating systems has taken computer usage full cycle from the open shop, job-at-a-time approach to the batched, closed shop environment, and back again to a flexible combination of on-line, time-shared execution coupled with batched production jobs. In similar manner, we have seen a pattern of independent computers, followed by complete centralization, and now distributed computers in a network approach. Needless to say, the extremes of open or closed shop operations did not represent a true picture of the needs of the real world. Although there were a number of special purpose systems, particularly command and control systems that utilized advanced operation system ideas ahead of their time, it has taken awhile for these operational concepts to evolve in the commercial world, and they are still evolving along with hardware and software developments.

The early days of job execution in the 1950's found things done on a single-program-at-a-time basis. A programmer personally, or through instructions to the computer operator, supervised program execution, directing all input/output routines with his own code, coding specific peripherals, and tying up the complete computer configuration.

In the interests of efficiency, the closed shop kept machine operations in "specialized" hands and thereby helped to increase throughput. However, job setup time, during which the computer was in an idle condition, still was a problem. This was true because of the lack of permanent auxiliary storage; all jobs had to be read singly and directly into the machine. Even with the use of magnetic tapes, there were delays in setup time and tape changes. The evolving operating systems were aimed at automating the sequence of job execution.

The relative slowness of peripheral input/ouput equipment; i.e., card readers, punches and printers, compared to magnetic tape, pointed to the next area of throughput efficiency. Thus, in the mid-1950's, early batch monitor systems (see Figure 2) were used as simple loaders to control the sequence of jobs and program processors read in from magnetic tapes.[24] These monitors were themselves self-loading from tape. Primitive "multiprogramming," the overlapping of program execution with input/output, was accomplished by off-line use of card-to-tape and tape-to-printer equipment.

The availability of magnetic tape made possible new benfits to ease the chores of the programmer. By standardizing input/output to system tapes, the use of common library input/output routines was encouraged. This sharing was particularly promoted with the advent of FORTRAN where such facility was conveniently made available. The use of magnetic tapes also escalated the subroutine library concept in a dynamic way: user programs could capitalize on the availability of library routines loaded from a library tape when the user program was loaded. These routines were useful only if flexible enough to be used anywhere in the object program. Finally, magnetic tapes fostered the overlay concept by allowing portions of a job, too large for the available core memory, to be brought into core when called by the executing program. This concept expanded the role of the loading function of these primitive monitors.

The impact on programming languages became considerable. The loading function and calling sequences, as well as availability of certain functions at execution time, required language development compatible with this operating system environment.

Hardware constraints on computer operations; i.e., core size, lack of sophisticated independent input/output, direct access secondary storage, etc., all tended to keep computer operations limited to one job at a time in the main frame. As long as this was true, there was little need for sophisticated accounting, security protection for files and hardware protection and control of machine resources. However, toward the end of the 1950's, the advent of independent input/output channels pointed to the use of multiprogramming concepts and concurrent usage of machine resources to increase throughput efficiency.

In order to take advantage of and service the new I/O hardware facilities, it became necessary to install a resident software package called an I/O or interrupt supervisor. On the 7090, there was IOCS and it provided I/O-CPU overlap and core buffering to keep the CPU busy. Service does not come free; one of the penalties of the I/O supervisor was the preemption of scarce core memory. Old programs had to be converted to use the new environment. All of this contributed to a somewhat negative attitude on the part of many of the early assembly language programmers toward monitor systems.

Whether the programmers liked it or not, it was necessary to preserve the integrity of resident routines. Thus, software methods for protection were adopted. Since it was possible to violate the resident routines and tables through erroneous I/O requests, it was appropriate to check the call requests for legitimacy prior to execution. I/O error conditions had to be properly handled so that object program execution could be gracefully terminated. Many a running system died because a user program branched into or stored data in the supervisor area and ran amok.

With the repertoire of execution time services provided by resident monitors, a new language appeared. This was for control of job execution (704 Monitor), handling such matters as peripheral device assignments, job accounting, job abort handling, and loading information. While job control languages have been the source of sophisticated processing procedures, they have also been the source of much waste of time and machine resources because of sensitivity to erroneous usage.

In the early 1960's, computer architecture took a turn toward "real time." This involved a real-time clock, the heavy use of interrupts, and the increased need to make the resident operating system inviolate. The addition of hardware protection, boundary registers, provided safeguards for sensitive areas of memory.

Although drums were available for fast secondary storage, their limited size and high cost inhibited wide use. Disk files provided high volume storage which was faster than tapes and less expensive than drums. Disk files made practical the use of resident monitor systems which could be overlaid from disk and also provided faster loading from the subroutine library.

Most significantly, multiprogramming techniques became highly practical in several ways, based on fast access drums and high capacity disks. The multiprogramming approaches involved staging peripheral input/output via disk, or spooling. This permitted faster program execution because all input and output occurred via disks rather than the slower card readers or printers. Another variation of this approach was the direct coupled system or "moonlight" system (e.g., 7094/1440) where the main computer performs the computation and the smaller computer controls the peripheral I/O, using a disk. Last, but not least, multiprogramming was applied to on-line interactive computer usage.

The pioneering general purpose time-sharing systems such as MIT's CTSS[25] and SDC's Q-32 TSS[26] highlighted the next round of facilities and responsibilities for modern operating systems that evolved during the latter part of the 1960's. Hardware protection was augmented to include Master-Slave modes of operation which enabled only the resident operating system or other privileged programs to execute I/O and certain control instructions. User or problem programs are trapped if any attempt is made to execute such privileged instructions. The operating system also had to provide access protection for centralized files.

With the shared concurrent use of system resources, such as core, disks, and I/O channels, the old wall clock method of computer usage accounting became obsolete.[27] Now it is important for the operating systems to account more specifically for the resources used, e.g., core memory, I/O, CPU time, disk storage and communication lines. This accounting capability has become very critical for proper operational management and configuration tuning.

The capability of relocating loader functions of operating systems increased with the loading of object programs via permanent secondary storage. The ATLAS computer in England played a great pioneering role in using secondary storage concepts.[28] Such hardware developments as memory maps for automatic relocation or virtual memory machines to eliminate the need for program overlay structures are currently impacting programming design. The concurrent usage of a program by several users in the time-sharing environment has given greater emphasis to potential savings from reentrant coding or pure procedures where all modifiable data or variable data is separated from the reentrant code. Job scheduling has become more intricate in terms of concocting priority schemes and scheduling algorithms for maximizing throughput or optimizing turn-around time for particular job classes (especially conversational processing in the time-sharing environment).

With the on-line multi-user system, the need for reliable operation became critical. As a result, recovery and restart facilities were added to operating systems. There is a great difference between rerunning a user job by the batch operator and having a large number of terminal users banging away at dead keyboards.

Communications handling by the operating system became more standard for batch work and for conversational terminals.[29] The marriage of data communications with data processing, among other benefits, permitted the effective implementation of centralized data bases, so critical to the needs of the modern business world. This in turn, has emphasized the need for separation of processing routines from data structures in programming design and development.

There are other practical benefits from the communications interface in terms of the network concept. The ARPA network, for example, is an indication of the linking of independent systems, many having "specialties." A user at a terminal can have access to pertinent data and processing capabilities other than his own.

The current trend of computer operations is leading away from restrictive batch production only toward a more flexible operational environment. Hands-on interactive facilities are of growing importance. The modern operating system also offers the in-between approach of conversational remote job entry which can exploit the effectiveness of both batch and interactive modes of operation. The evolution of operating systems has been one of cumulative growth and expansion of facilities along with control responsibilities. This growth can be traced in Figure 2. Future growth appears to be limited to consolidation and refinement of the current scope of functions with major emphasis upon hardware/software architectural synergism.

ON-LINE SYSTEMS PROGRAMMING

From the earliest days of development of the computer, consideration was given to electronic systems in which the computer was an imbedded piece, usually the control element. The early systems were those for the military, such as fire control and air defense systems.[30,31] However, it was not until the early 1960's that on-line systems* became sufficiently active to have a definable

* We use the term "on-line systems" to refer to all systems in which the computer is attached to instrumentation from which it receives (and gives) signals asynchronous to its operation. This terminology, we find, is superior to "real time." "Time-sharing" systems refer to those on-line systems in which many users are "simultaneously" using the machine.

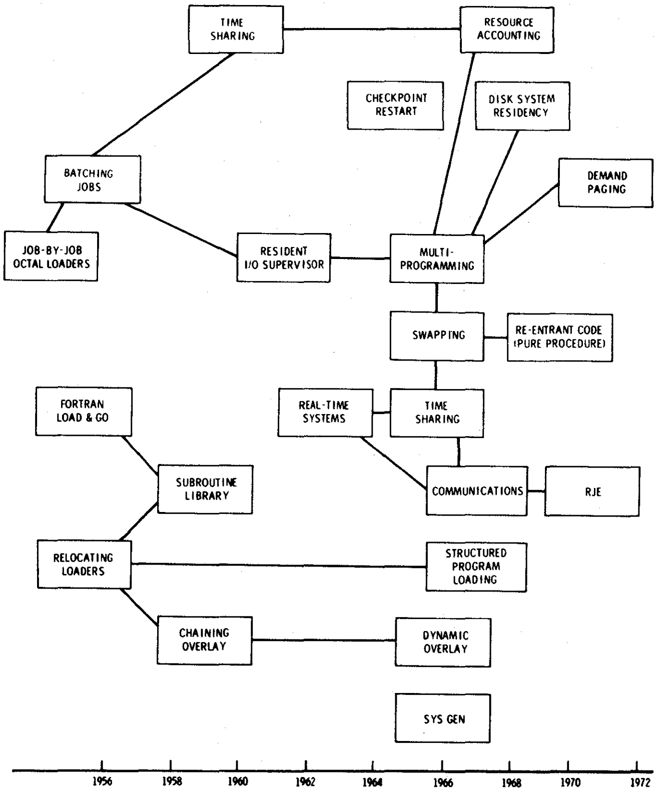


Figure 2—Evolution of operating system functions

programming structure.[32] By the late 1950's, the interrupt feature was "invented" and appeared on large scale computers such as IBM's Stretch and Univac's LARC, introduced in the early 1960's. Programmers were increasing their understanding of the relationship to interconnecting instrumentation and were becoming better able to cope with it. In the early 1950's, the information was given to the card reader and the program "waited" until the entire card was punched. By the late 1950's, with the introduction of computers such as the IBM 709, the programmer was doing useful computation between punching of successive card rows. The interrupt feature was important in cases such as this; it allowed the program to continue the processing, ignoring the operating of the external devices (and not having repeatedly to test its status), and wait confidently for the interrupt to occur to cause control to be transferred to the interrupt handling routine.

By the late 1950's, the subject of "multiprogramming" or "parallel programming" was being investigated.[33] This approach was aimed at better machine efficiencies through the overlap of various machine functions hitherto done purely sequentially. Computation could well proceed while memory cycles were "stolen" to complete an input-output process. Also, the approach of tying computers together ("multicomputers")[34,35] was receiving increasing attention. These techniques, aimed at more efficient machine operation in increasingly complex application environments, required new programming solutions. Control programs were needed to keep all the units operating efficiently in concert.

An important subset of on-line systems is the computer/communications system. The development of these systems has been traced, and the system defined elsewhere.[36] The development of particular functions such as network control, message handling and line control has resulted in an important programming specialty, and one of growing importance.

The beating heart of the on-line system is the "Master Control Program," sometimes referred to as the "executive." This Master Control Program was a unique, definable entity for it had no counterpart in general purpose computing. The MCP performed the functions of scheduling, controlling, synchronizing and monitoring the entire process. It took its place alongside of and made use of utility programs (data moving, memory controlling), console programs (the data formatting, controlling operator logic), and application programs.

By 1962, the subject had developed to a point where a definitive paper[36] could be written on the subject. Subjects which were now clearly understood, and hence amenable for further development and more extensive application included: man/machine interaction via consoles, master control program design, multicomputer and "graceful degradation" aspects, and comprehensive interrupt handling.

General purpose operating systems and their application programs undoubtedly benefited from the on-line systems development that started with the military systems. Time-sharing, interactive systems, communication systems, the modern operating systems, and the world of minicomputer applications can be said to have these on-line and multiprogramming concepts as direct antecedents. If one looks at the character of the on-line systems (mostly military) developed prior to 1962, one sees that these modern day systems perform many of the functions first performed and provided in these earlier systems.

EVOLUTION OF SOFTWARE PRODUCTS*

The notion of software as a product came late. This was probably so because of the program-sharing spirit of early computer pioneers. Most of these people were

* Sometimes called "program products."

of the academic or scientific viewpoint. They believed in immediate publication and full disclosure of all developments. Further, there was no thought that software could become a commercially important product. The tradition of freely shared software was fostered by organizations such as SHARE, USE, and GUIDE.

It is hard to name the first significant instances of software sold for money. Codes for linear programming were developed by joint ventures in the late 1950's. They may have been the first proprietary software sold in the United States. Linear programming is important to the work of the petroleum industry. Companies in this industry developed linear programming packages either singly or in coalition. Some commercial software and service companies developed their own proprietary methods and sold the services of these, though they rarely sold the software itself, since the demand was insufficient.

The mid-1960's was a period of controversy about software products. Some spokesmen proclaimed the bright future of software products as early as 1962, while other pundits as recently as 1968 said there could be no such thing. Much of the confusion about software products can be attributed to a lack of understanding about what kinds of software could be "productized." General-purpose "tools," which can be utilized as-is by users, make the ideal software product. Application packages, on the other hand, particularly those which are data base dependent, will usually require "tailoring" before they are of practical value to individual users. This is true because of the lack of standards in the specific application areas and the dynamics of real world needs.

The economic leverage of the software product is too great to be ignored forever. In view of ever-increasing machine capability and dramatic increase in computer cost effectiveness, the path to ever-increasing software complexity was wide open and lined with roses. Sooner or later the world had to face the high cost of completely home-grown, custom software. The distressing shortage of highly competent software system designers and system programmers did nothing to improve the picture.

The resulting flourish of software entrepreneurs blossomed remarkably rapidly. Software companies sprang up everywhere. They were, in many instances, encouraged by the view that the software product has miniscule manufacturing costs, except at first item delivery. The price bidding could be against the cost of a purportedly equivalent in-house product, and the software supplier might indeed come out looking pretty good.

Some of the brave hopefuls among the many small and large software companies did not survive to see their planned products delivered. Some of them had badly underestimated the development costs of the first delivery item. Others presumed that no manufacturing costs meant no marketing costs. But the most consequential and glaring mistake was the significant underestimation of maintenance and improvement costs and general post-sale customer support.

By the mid-1960's, few if any data processing groups would consider in-house development of a compiler. Also, by that time the idea of developing any significant piece of systems software or programming tool on a one-of-a-kind basis was becoming suspect.

Software products became real for all the world to see when IBM, largely in response to anti-trust pressures, announced unbundling. Remaining doubts about the reality of software as a product quickly disappeared. The 1972 Datamation Industry Directory lists 41 pages of them in 119 categories.

About the time of IBM unbundling, there were many predictions about software products, some rather ill advised. It became a mistaken belief, particularly among starry-eyed startup investors, that any piece of software which purportedly worked was "a software product." There were some bad financial losses and disappointments and a pervading mistrust of software products for a variety of psychological reasons. Fortunately, both time and need are overcoming the mistakes of the past.

While the patent situation is still unclear with regard to software, lack of patent protection will not be a deterrent to future developments of software products. The trade secret body of law, and binding contracts with customers and with employees, afford ample protection for the software products supplier.

It is interesting to note the differences between software products and hardware products. Design and development is expensive in both areas of business. Maintenance is highly decentralized with hardware, but much more centralized with software, since the change need only be made once, then copied to all users. Software has essentially no manufacturing costs. Marketing costs are high for both hardware and software. As a percentage of revenue, marketing costs are currently higher for software than for hardware.

FUTURE TRENDS

The central part of this paper traced the developments of programming during the modern short twenty year history of modern computing. On the theory that we learn from the past in order to better equip for the future, it seems appropriate in the light of the generated historical perspective to comment on the future of programming. These comments take the form of opinions on the current state of programming matters and predictions for the future.

The day of programming languages being developed by the hardware manufacturer passed in the late 1950's or early 1960's. COBOL, the last language to be developed which has in any sense gained universal acceptance (in the business world), is now more than ten years old. FORTRAN, still going strong, is even older. The last attempt at a more "universal" language, PL/1, has been something less than a spectacular success. There is nothing on the horizon to suggest the development of a single comprehensive language for all scientific and business applications.

The programming profession has apparently come to realize that there is no such thing as a "universal language." It has probably also come to realize that there is a fundamental conflict between the power of a language by a wide class of applications on one hand and the universality of that language on the other. Languages in the framework of a burgeoning set of application "areas" cannot be all things to all users.

It is appropriate to distinguish between a computer-based application which processes a specific set of input parameters and produces a specific set of outputs and an application "area" such as numerical control, photocomposition, computer-aided instruction, etc., where a whole class of problems including unique terminology and functions, must be dealt with in a flexible manner. The latter requires a special "problem-oriented" *language*, while the former needs a specific application *program*.

The software era that we see the industry entering is a phase of maturation that reflects both understanding of real world needs and technological advances. Software will be separable into three major areas:

- Tools for the development of production applications; e.g., language compilers, file menagement systems, utility packages, etc.;
- Application programs for the direct use of *end users* (or industrial processes). This will also include more general purpose language systems for simple *ad hoc* inquiry and computation; or,
- Operation control programs—operating systems that control execution of user programs in an increasingly complex machine environment.

In the area of scientific applications, the major advances appear to be the incorporation of those computational and manipulative functions (such as array manipulation) that the computer makes so convenient into language notation, as has been done with APL. However, the scientific community has been reasonably satisfied in terms of language needs for defining their problems. "Problem-oriented" languages will continue to be developed as well-defined application areas with unique requirements providing some basis of standardization.

The pressing needs of the business world, however, which are more pervasive and have been more neglected than the needs of the scientific community, will further help to establish the concept of data structures that can be independent of program structures. Data and file descriptions will permit data access and manipulation to be accomplished by application programs dynamically at execution time. There will also be an expansion of facilities for the proper management and control of data within an operational system.

It would appear that in the foreseeable future, language development for business data processing will consist of two mainstreams . . . COBOL-oriented preprocessors (including shorthand and optimizer preprocessors) and the file management system approach. The COBOL generator approach seems to be consistent with evolving techniques throughout the history of programming: layers of capability are successively added to the machine in the forms of software systems. The operating system relies heavily upon assembly language. The COBOL layer is developed upon this combination of hardware and software. Shorthand languages are developed to facilitate the preparation of COBOL programs. It is clear that with the need for stability and continuity in business applications, the preprocessor approach seems destined to become increasingly important in the future.

In earlier sections of this paper, the reader has seen how the file management approach evolved from the early report generator systems. It seems clear that these generalized data management systems represent the only possible claim to a language (or a type of language) to compete with COBOL. In the realm of *ad hoc* (one-time) retrieval, these systems have a clear-cut superiority over COBOL, and all application areas have a need for such a capability. These packages are becoming increasingly powerful and increasingly used. The trend will undoubtedly continue. There will be a profusion of enhancements encompassing complete data base needs and resulting in systems with faster programming execution and manifold options with which to tailor systems to the particular application needs of the user. COBOL achieved some degree of universal acceptance because it was developed during the days before software was considered a business in its own right and because it had the power of the Department of Defense backing it; the file (or data base) management systems being developed today are proprietary items being developed by private industry and, although no single version among them is likely to achieve industry-wide endorsement, as a class, data management systems are likely the wave of the future.

Programming languages for application development will consist of not merely the compiler or code generator, but a complete language-compatible set of tools for debugging, maintenance, documentation, etc. Thus, we must think of a language "system" which starts with application design needs through production operational requirements.

The concept of applying "layers" of language will become most significant in the area of applications. By definition, every application program creates a new king of language; i.e., its data input format. This is true for highly parameterized inputs and for more flexible, *ad hoc* query and computational languages for specific applications. The preprocessing approach is most effective when a highly interpretive mode is needed for an interactive phase of operation or for very user-oriented, automatic code generation, *and* the "under layer" language is an efficient standard. Compilers are frequently preprocessors for assembly language; application preprocessors for compilers and interpreters, particularly for business use, are coming into wider usage.

The need for simplifying the programming effort will promote the use of automatic and default programming tools. This has proved particularly successful in business applications where common needs are stabilizing and coding ingenuity does not produce much practical payoff. It is not sufficient to provide tools that require sophisticated expertise to avoid mistakes and misuse. It must be there if necessary, but the common need can be handled in a more simple, automatic fashion. The file management and report generator packages have notably demonstrated the effectiveness of this approach.

The development of comprehensive operating systems is relatively recent, considering that the first such "running operating systems" really came into being with the third generation computers in the mid-1960's. We have seen only the beginning of what will undoubtedly be an exceedingly important development in the years to come. The extraordinarily high speed of the modern computer, the relatively low cost of high speed memory, and in general, the dramatic increase in cost effectiveness, means that complex layers of "machine" capability (layers of software, actually) will be developed, making the job easier for the programmer at the application or the compiler level. The operating system developer in reality is building a new machine on the basis of stored logic principles starting from the standard machine instructions. The trend is clearly evident over a decade. There is no question that it will continue.

The role of operating systems has become significant enough to warrant the concern over "operating system independence." It is obvious from a practical standpoint that computer operations must be sustained for the real world production environment. Not only must there be no drastic interface changes, but there must be graceful, upward compatibility with new facilities in the areas of communications, data structures, storage devices, etc.

It is apparent that the general purpose operating system will allow the user installations a wide range of operational modes ranging from batch, remote batch, to highly interactive as well as transaction-oriented, online usage. The distributed network of computer processing will also expand remote access and the concept of computer-based services.

Because of the increased operational usage and the varied response requirements, the proper management of configuration resources will become most significant. Thus, instrumentation for evaluating throughput performance, coupled with controls for dynamically allocating priorities and resources, will assist in the understanding and the effective management control of day-to-day operations.

The complexity of operating system facilities which has caused the command language or job control language to proliferate, will cause more user-oriented, automatic generation of such operating requirements to be "buried" with the applications program.

The last ten years have also seen a marked increase in the appreciation of cost of complex software systems. This is leading, in turn, to a most important trend which does not seem to be clearly recognized. In the past, whenever the programmer received a new machine, his immediate and first thought was to reprogram his application so that it would operate on the new machine. Now that many applications are more stabilized and the high cost of programming systems more clearly recognized, the hardware will be brought with increasing frequency to the software rather than vice versa. This means that the emulation by one machine (presumably the more technologically advanced one) of another machine will become increasingly important. This is mainly accomplished through microprogramming and stored logic techniques. We have seen this approach taken in the important commercial arena with the third generation computers. We have seen the idea advanced through hardware with the use of read-only memories and other microprogramming techniques. As long as cost effectiveness of hardware increases at the rate it has in recent years and cost effectiveness of software approaches improves much less rapidly, there will be increasing pressures to bring the hardware to the software.

Hardware approaches to make the programmer's life easier will proliferate. In the mid-1960's, we saw the first machines of commercial importance expressly designed for time-sharing. These systems included hardware and software implemented capabilities which allowed the programmer to prepare his programs as if he were the only user and as if he had unlimited memory. The latter involved so-called "paging" and virtual memory hardware facilities. The advent of these systems underscored the importance and acceptance of resource-sharing techniques in the industry. It is likely that by 1980, even with bigger, cheaper core memories, nearly all new machines will have these capabilities and further relieve the applications programmer of the concern for core storage limitations.

Application packages as proprietary software will become an important commercial area. Until now, the customer of (or prospect for) an application package has been unwilling to accept the system except in the exact form he conceives it, all the way from computing algorithms to the form of output reports. On the other hand, the supplier community has not developed techniques which allow the creation of flexible, automatically tailored application software. This gap will narrow; the customer community will become more tolerant in view of the economics of the situation, and the suppliers will design and build more flexible software.

Utility packages (sorts, compilers, data management systems, etc.) will continue to grow as important proprietary software.

CONCLUDING OBSERVATIONS

We cannot let the opportunity pass to wax philosophic about programming.

Senior observers of the last twenty-five years of programming development all agree on one point: it is a continuing shock and surprise to realize anew the span of time between the development of an idea and its widespread, practical use. Old-timers recall how slow the programming professional was in 1958 to embrace FORTRAN. In 1964, why weren't almost all business applications being programmed in COBOL? The lack of fully checked-out, fully compatible systems is not the answer, although such system problems contributed in a minor way. The answer is that there seems to be an inherent conservatism within the professional programmer working in business and industry. Is he subconsciously increasing his job security? Is the spectrum of progressiveness and inventiveness extraordinarily wide within the programming profession? Is the lack of proper education at fault? Finally, should we blame the dominant computer manufacturers for *de facto* control of progress?

But let there be no doubt about the fact that change has occurred.* The developments recorded on these pages are mute but uncompromising evidence of that. Nevertheless, it is disappointing that, in the estimate of most, programming cost effectiveness improves slowly, probably 5 percent per year compared to a whopping estimated 25 percent for hardware. Perhaps this is due to immutable, intrinsic characteristics. But it may be a result of the fact that the data processing industry has placed true programming at too low a professional level. The lack of attempts within the profession to structure the field and organize and discipline programming activity are also contributing factors of a more minor nature.

REFERENCES

1 J A POSTLEY
Computers and people
McGraw-Hill New York 1960

2 W S MELAHN
Description of a cooperative venture in the production of an automatic coding system
Journal of the ACM Vol 3 No 4 November 1951

3 S ROSEN
Programming systems and languages
McGraw-Hill 1967

4 M V WILKES D J WHEELER S GILL
The preparation of programs for an electronic digital computer
Addison-Wesley Press Reading Mass 1951

5 G M HOPPER
The education of a computer
Proceedings of the Conference of the ACM Pittsburgh 1952

6 JEAN E SAMMET
Programming languages: history and fundamentals
Prentice-Hall Inc Englewood Cliffs NJ 1969 pp 12–13

7 J W BACKUS
The IBM 701 speedcoding system
Journal of the ACM Vol 1 January 1954 p 4

8 R R EVERETT
The Whirlwind I computer
Electronic Engineering 71 August 1952 pp 681–686

9 S GREENWALD R C HOUETER
S N ALEXANDER
SEAC
Proceedings of the IRE Vol 41 (October 1953) pp 1300-1313

10 *Commercially-available general-purpose electronic digital computers of moderate price*
Proceedings of the Symposium of the Navy Mathematical Computing Advisory Panel and the Office of Naval Research Washington DC May 14 1952

11 M V WILKES
The best way to design an automatic calculating machine
Manchester U Computer Inaugural Conference 1951

* The twenty-fifth anniversary edition of the Communications of the Association of Computing Machinery[38] was published too late for inclusion as a major source of references for this paper. However, it has many valuable articles which relate to the topic of software history and trends.

12 R F ROSIN
Contemporary concepts of microprogramming and emulation
Computing Surveys ACM Vol 1 No 4 Dec 1969

13 *COBOL: initial specifications for a common business oriented language*
Department of Defense US Government Printing Office Washington DC April 1960

14 R BEMER
A view of the history of COBOL
Honeywell Computer Journal Vol 5 No 3 pp 130-135 1971

15 J STRONG et al
The problem of programming communication with changing machines: a proposed solution
Comm ACM Vol 1 No 8 1958

16 SAMMET Op Cit p 5

17 A J PERLIS K SAMUELSON (for the committee)
Preliminary report—international algebraic language
Comm ACM Vol 1 No 12 Dec 1958

18 E L WILEY et al
A critical discussion of COBOL
Annual Review in Automatic Programming Vol 2
Pergamon Press New York 1961 pp 293-304

19 *IBM system/360 operating system: PL/1 language specifications*
IBM Corp C 28-6571-0 Data Processing Div White Plains NY 1965

20 J C SHAW
JOSS: A designer's view of an experimental on-line computing system
Proc FJCC 1964

21 J G KEMENY T E KARTZ
BASIC
Dartmouth College Computation Center June 1961

22 K E IVERSON
A programming language
John Wiley & Sons, New York 1962

23 J A POSTLEY
The MARK IV system
Datamation January 1968

24 C L BAKER
The PACT coding system for the IBM type 701
Journal of the ACM Vol 3 No 4 October 1956 pp 272-78

25 F J CORBATO et al
The compatible time-sharing system, a programmer's guide
MIT Press Cambridge Mass 1963

26 J I SCHWARTZ E COFFMAN C WEISSMAN
A general-purpose time-sharing system
Proceedings of the Spring Joint Computer Conference 1964

27 A M ROSENBERG
Computer usage accounting for generalized time-sharing systems
Communications of the ACM Vol 7 No 5 1967

28 T KILBURN B G EDWARDS M J LANIGAN F H SUMNER
One-level storage system
IRE Transactions April 1962

29 A M ROSENBERG
Group communications in on-line systems
Proceedings of On-Line Computing Systems Symposium UCLA/Informatics 1965

30 E H GOODMAN
Sage
Computing News Vol 6b Nos 15–17 Aug through Sept 1958

31 W F BAUER W L FRANK
Doddac—An integrated system for data processing, interrogation, and display
Proceedings of the EJCC December 1961

32 W F BAUER
On-line systems—Their characteristics and motivation
On-Line Computing Systems (Proceedings of the Symposium) America Data Processing Inc Detroit 1965 pp 14-24

33 S GILL
Parallel programming
Journal of the British Computer Society 1957

34 R PERKINS W C McGEE
Programmed control of multi-computer systems
Proceedings of the IFIP Congress—1962 Munich Aug-Sept 1962

35 W F BAUER
Why Multi-Computers?
Datamation August 1962

36 W F BAUER
Computer communication systems: patterns and prospects
(Proceedings of the Symposium on Computers and Communications) Toward a Computer Utility Prentice-Hall Englewood Cliffs New Jersey 1968

37 W L GORDON G L STOCK
Programming on-line systems
Datamation September 1962

38 *Communications of the Association of Computing Machinery*
Vol 15 No 7 July 1972

NASDAQ—The evolution of automation in OTC trading

by GEORGE E. BELTZ

Bunker Ramo Corporation
Trumbull, Connecticut

THE TRADING PROCESS

There is a fundamental difference between trading in listed securities and OTC securities that has spurred the need for automation in the OTC industry. Listed securities are, in general, traded only on the floor of an exchange—in essence, a central market place in which a single BID and ASK price are set, and distributed via a ticker and stock quotation systems. OTC securities, on the other hand, are traded in a vast decentralized market place, by thousands of broker dealers scattered nationwide.

The evidence of problems

In the mid 1960's, it was often difficult and sometimes impossible to obtain up-to-the-minute, accurate quotations on OTC securities. The only means available to a private investor for the checking of his OTC investment was either to consult the OTC page of his newspaper, the problem being that newspapers would often have only 1,200 or less issues, and the quotations would be at least one (1) day old, or to call his customer's man, who in turn would either consult the pink sheets, which contained quotation information at least one (1) day old, or have the firm's trading department find a market maker willing to buy or sell—in any event, it was a very time consuming and inaccurate approach. What was happening was the public was not adequately being served in that the quotation information was often old and many times misleading, and the requirement that an OTC trade be executed only after obtaining three (3) quotations was often bypassed.

Market making firms and OTC trading departments were also having difficulties in that large staffs were required to man the phones, trying to obtain current quotations. An additional problem was that only a very small number of the requests for quotations actively resulted in a trade, and to add insult to injury because of the numerous phone calls requesting quotation, a firm's telephone lines were often tied up and legitimate trade requests were many times unable to get processed.

Addressing the problem

It was in this environment that in 1966, Bunker Ramo submitted a proposal to the NASD outlining the concept of an on-line nationwide quotation system displaying up-to-the-second quotation information. The NASD, upon receiving this proposal, retained consulting services to help study the feasibility of the approach, and if deemed feasible, to prepare a Request for Proposal. The RFP was submitted to approximately ten (10) firms and in December of 1968, Bunker Ramo was chosen by the NASD to be the NASDAQ System operator.

In the middle of 1968, however, concurrently with early NASDAQ developments, the National Security Traders Association selected Bunker Ramo to implement a quick and inexpensive solution to the OTC quotation problem. This led to the implementation, in February of 1969, of the STAQ System, an acronym for Security Traders Association Quotation System.

The STAQ System concept was to designate one market making firm for each OTC security and allow that firm's market maker to maintain a current quotation for the security. Market makers were asked to submit their quotations on a 15-minute schedule, but because of lack of incentive and for many other reasons many times, quotations were not kept current, and therefore, competing firms would not conduct business at the STAQ quotation price. While not the real answer to the OTC problem, the STAQ System did provide an improvement over the prior "Quotation-by-Telephone" approach.

The main difference between STAQ and NASDAQ is that under the STAQ concept only one firm represents a security, whereas under the NASDAQ concept all authorized market making firms are allowed to enter quotations in a security and make these quotations instantaneously available to all OTC traders. This concept of the NASDAQ System not only forces market makers to keep their quotes current and honor their prices, but also has resulted in better trade executions.

IMPLEMENTING THE NASDAQ SYSTEM

In December of 1968, after the signing of the agreement with the NASD, Bunker Ramo began forming a design team to implement the NASDAQ System. Although the contract contained basic requirements, there were many unresolved details that had to be explored and checked. It was a new concept, and no one knew exactly how the subscribers would use it, or what effect the system would have on doing business in the OTC market. After four (4) months, representing 40 man-months of countless meetings with the NASD and potential customers, the Functional Specifications for the system were completed. After an elasped time of 25 months and the expenditure of approximately 100 man-years of system and programming and hardware design effort, the NASDAQ System went fully operational, serving an initial population of over 1,100 terminals in over 700 offices nationwide.

The project group

A Project Group, within Engineering, was formed and assigned overall responsibility for project planning, scheduling, coordination, and implementation. The group consisted of a Project Director, an Administrative Assistant, a Systems Manager, a Programming Manager, and their staffs. The group functioned as a team with a common objective. Systems and programming personnel worked hand-in-hand to analyze each task and establish approaches. There was not the formality of Systems, alone, preparing a design specification and handing it over to Programming for implementation.

Liaison with the NASD was handled through the Systems Group, which also acted as arbitrator and final decision maker on internal discussions.

A 200-page design specification detailing system inputs, processing actions and rules, and system outputs was the result of some 40 man-months of team effort working with the NASD. This document, with the concurrence of the NASD, became the bible for the design effort which followed.

The Project Group handled, in addition to Systems and Programming, vendor evaluation, processor selection and acceptance testing, site layout, power fallback selection and checkout, broker surveys, modem evaluation and selection, specifications, and checkout of Bunker Ramo designed terminal equipment and the coordination of Manufacturing and Field efforts.

Most systems personnel had previous involvement in other Bunker Ramo on-line real-time system implementation and had background in Logic Design, circuit design, communication techniques, and programming.

Programming was subdivided into four groups: On-Line, Off-Line, File Maintenance, and Recovery. Key senior people with previous on-line real-time involvement with Bunker Ramo systems headed up these groups working under the Supervisor and a Programming Manager. Again, this was a group which worked together to establish approaches, and iron out problems.

When the Project Group was formed in December, 1968, it consisted of seven Systems, and four Programming personnel. This grew to a peak of ten Systems and 35 Programming personnel. Console Operators and Keypunch Operators were brought on, starting in October of 1969, and totaled ten at cutover.

Monitoring the Project

A two-year implementation schedule was established and laid out on a milestone chart, which was reviewed first on a monthly basis; and at crucial points, on a weekly basis. Review sessions included Department Vice-Presidents, Group Managers, and Directors. Weekly progress reports were prepared for management on each aspect of current activity. Each major activity was assigned a category, which in turn, was further subdivided into items. Labor, material and other disbursements were booked by category and item, and a detailed report, by cost center against this category/item list was prepared monthly by Accounting.

Budget preparation and monitoring was a responsibility of the Project Group.

SYSTEM SERVICES

In designing the NASDAQ System, we have therefore, attempted to address the problems that have been basic to the Over-the-Counter Market since its beginning. The main problem is one of visibility: (1) by the market maker of his competition and of their cur-

Level 3	Level 2	Level 1
Market Maker	*Traders*	*Distributors*
News Requests	News Requests	RBA Prices
Quote Requests	Quote Requests	Indices
Indices Requests	Indices Requests	
Volume Requests		
Quote Requests		
Volume Updates		

Figure 1—NASDAQ services
Levels 1, 2, and 3

rent prices; (2) by the retail trader of those market makers willing to buy or sell securities, and at what price; and, (3) by the general public as to security prices, trading volumes, and stock indices. Last, but certainly not least, visibility is important to the NASD, the self-regulating body of the OTC industry, to enable it to supervise the activities of its membership in security dealings.

The NASDAQ System provides the following services:

Level 3 service to Market Makers: (Figure 1)
a. Quote Request (Bid or Ask)
b. News Request
c. Indices Request
d. Quote Update
e. Quote Withdraw
f. Quote Re-Open
g. Quote Close
h. Volume Update and Read

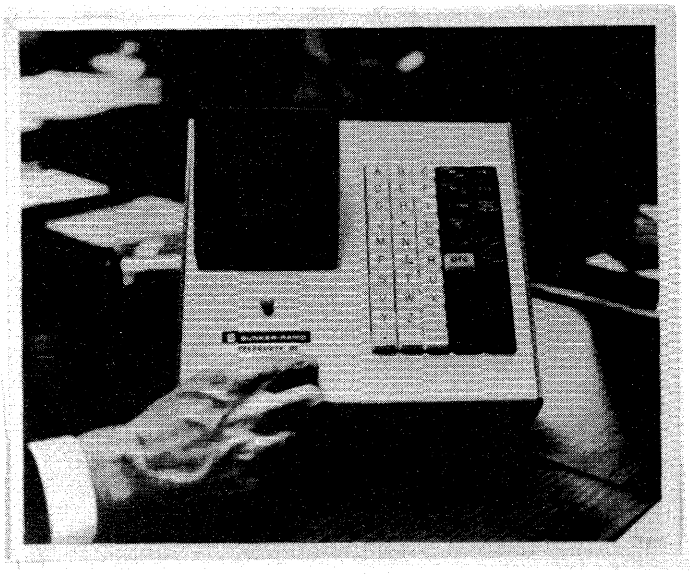


Figure 2—Bunker Ramo Level 1 terminal

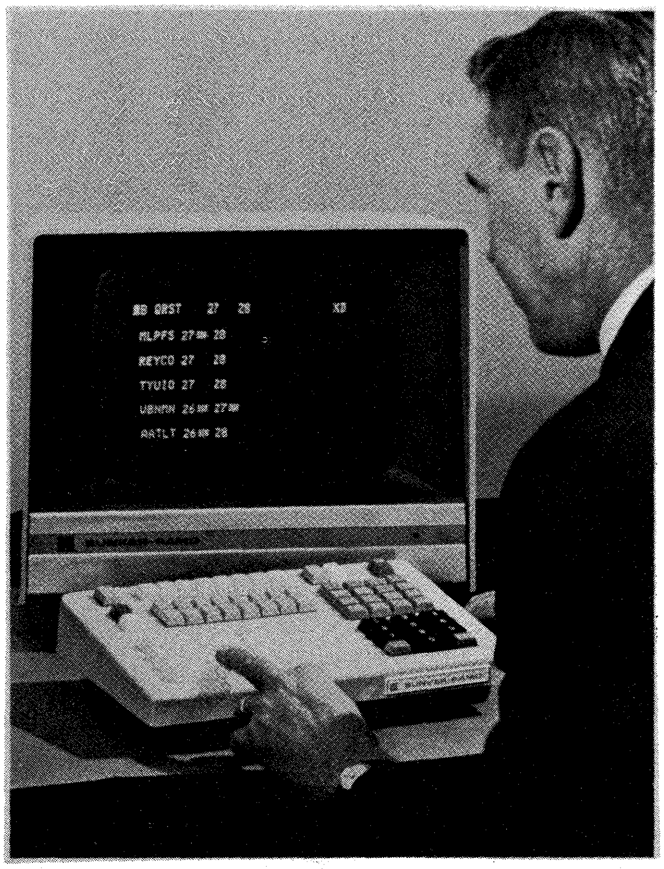


Figure 3—NASDAQ Level 2 and Level 3 terminal

Level 2 service to Traders:
a. Quote Request (Bid or Ask)
b. News Request
c. Indices Request

Level 1 Service to Quotation Distributors:

Representative Bid/Ask (RBA) prices, as calculated by the system based on all market makers in each security are fed, as they change, to Bunker Ramo, Scantlin, and Ultronics, the three (3) major quotation distribution vendors. Included also are indices calculated at five (5) minute intervals.

Level 1 service is obtained typically through a Bunker Ramo Telequote III terminal, (Figure

- Representative Bid/Ask Prices
- Indices
- Volume
- Daily Market Summaries
- Recaps (Weekly, Monthly, Yearly)

Figure 4—NASDAQ Services
Newspaper/Newswire

2). Level 2 and Level 3 service is supplied only through NASDAQ terminals (Figure 3).

Services to Newspaper and Newswire Firms: (Figure 4)

Hourly transmissions to the newswires, AP and UPI, carry the latest RBA prices on all securities along with indices. Day end tramsmissions include volume on each security, based on the Market Maker reports.

Newspapers are provided with hard copy print out of the same information supplied to Newswire Services, at the New York Concentrator.

Not to be overlooked are the Supervisory Terminals at the NYC and Washington, D.C. offices of the NASD. From these terminals, the NASD has supervisory control of the system with full control over the market makers and securities in the system, and all aspects of system operation.

The system has many surveillance features built in, and daily generates a report on broker activities measured against various parameters. This report is transmitted to the NASD and printed in their office each morning prior to commencement of trading.

Weekly, Monthly, Yearly, and Special reports are all generated by the system, based on data captured daily, as each and every call entering the system is logged along with all internally generated data, including RBA prices and indices.

SOME ASPECTS OF SYSTEM PLANNING

Since an OTC quotation service was only the first phase of automation planned by the NASD, the processor selection and the system design had to be geared to both present and future needs. In the more immediate future was the plan to include Trade Reporting, Comparison, and possibly some phases of clearing. Phase I traffic handling and response time requirements were closely reviewed. Initial RFP requirements were stated as requiring the handling of approximately 29 calls per second, four (4) to five (5) years out into system operation; these objectives were drastically revised in the early stages of system specification, to call for handling approximately 200 calls per second in the second and third years of operation, with plans for increasing traffic handling capacity beyond that point.

A review of existing large scale processors, with emphasis on multi-programming, multi-processing, recovery capabilities, communication handling, fast access storage and reliability led to the selection of the Univac 1108 as the processor for the Central Processing Complex.

Since the system must serve brokers nationwide, communication line costs and communication polling loads were factors that were foremost in our consideration.

An analysis of communication line costs for the expected customer distribution indicated the need for remote concentrator points to serve customers in a region and to feed a central site via high speed line facilities.

Future prospects of after-hour batch operations at strategic centers, scattered nationwide, along with the need to de-centralize the communication polling load, led to the selection of a G.P. machine to serve as the remote Store and Forward concentrator. A Honeywell DDP-516, coupled with Bunker Ramo's own design communication front end, provided the capability to handle up to 64 polled full duplex synchronous, or asynchronous regional subscriber lines, and two (2) full duplex, concurrently operative trunk curcuits, at speeds up to 50,000 bits per second.

Subscriber equipment, including CRT's and keyboards designed to meet the users needs, along with terminal control units capable of handling up to 24 terminals and a variety of peripheral devices and options had to be designed, built, and installed. While this might seem a sufficiently difficult task to tackle, we had to include concentrator site selection preparation and installation, along with the design and construction of a new Computer Center, including a fallback power source.

The system, now officially in operation since February 8, 1971, consists of a Central Processing Complex at Trumbull, Connecticut, housing two (2) Univac 1108's working under Exec-8 in a multi-processing, multiprogramming environment. The hardware complex (Figure 5) consists of two (2) processors, three (3) 65K word core banks, one (1) of which is for fallback, two (2) dual access drum subsystems each handling two (2) 432 and two (2) 1782 drums, two (2) dual access tape subsystems with eight (8) tape drives, two (2) communication interface subsystems, and two (2) 9300 processors, each supporting a card reader, card punch, and line printer.

The hardware and recovery techniques were configured to insure that the system met the required reliability objectives, which included a maximum down time of ten (10) minutes per week and three (3) hours per year. To further protect against outages due to commercial power failures, or brown outs, the Center

Figure 5—NASDAQ data center

houses a 400 KVA Uninterruptible Power System with battery interim fallback, supported by a 1,000 KW turbine driven generator, fed from a 20,000 gallon supply of fuel oil, allowing the Center to be self-sufficient for nearly two (2) weeks without refueling.

Connecting the Center to the system's four (4) store-and-forward concentrator locations in a network of high speed, full duplex data circuits ranging from 7,200 bits per second to 50,000 bits per second, (Figure 6). Two (2) circuits connect the Center to each concentrator location. Traffic is handled concurrently on both circuits to utilize their available capacity and continually monitor their quality. Software and hardware techniques are used to monitor line faults and error rate levels. In the event of trouble occurring on one (1) of the two (2) circuits feeding a concentrator, all traffic is routed to the good circuit and, if necessary, a backup line is dialed up until full service is restored. Trunk line speeds were increased from 4,800 bits per second to 7,200 bits per second and will continue to be increased to insure sufficient capacity, even on a single line during fallback.

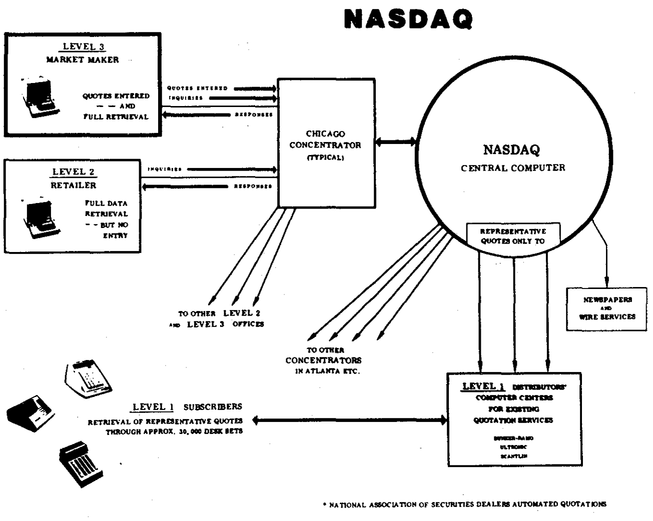


Figure 6—NASDAQ communications network

Concentrator sites are equipped with multiple sets of hardware, all actively handling a share of the communication load, but configured to allow switching of regional lines from a down machine to an active unit with a negligible transfer time. Concentrators may be loaded either locally from paper tape, or remotely via communication circuits from the 1108 complex as a result of a load request, thus minimizing recovery times. Concentrators act as store and forward units at present, but can be upgraded to Data Base concentrators in the future to dead end quote request traffic from subscribers, which is now running 20 times more than update traffic. This would relieve a significant load from the CPU, allowing it to take on other tasks.

MEETING THE OBJECTIVES

Let us consider some factors that have helped the NASDAQ System meet its objectives:

1. Prior to cutover, the system was subjected to a stringent functional and traffic test, both automated, wherein the system was driven from without, as opposed to simulated internal testing.

   An extensive scripting effort produced some 12,000 query response frames encompassing the full range of data variations, valid and invalid, which live operation could bring.

   A separate test processor complex simulating broker's hardware was connected in normal manner to the NASDAQ System via an operating concentrator. Calls were manually entered through the system and if handled properly, were logged on magnetic tape by the test complex. Design problems found were corrected in the process. When completed, we had a tape containing queries and proper responses, which could be played back at any time through the system to validate system integrity after new assemblies or program changes. The test complex, when run, verified all responses and printed out any deviations. This approach allowed us to simulate several days of system activity and to verify not only the on-line actions, but also the after-market report generation, based on a known set of inputs.

Also designed were special traffic generator programs for a separate test processor complex. The output of this complex was 32 communication lines, which were connected to a normal concentrator's regional line interface, thus simulating customer activity on all lines, thereby checking out the concentrator programs and the traffic handling capability of the concentrator and the Central Site Complex.

One additional set of programs was designed to replace the normal application program, in any concentrator, to allow it to generate background traffic into the system when conducting response time tests. In this testing phase, some 20 operators, in a simulated broker's complex, inputted calls into the system against background traffic generated by all remote concentrators. Response times were automatically registered on special counters attached to each operator's terminal.

2. Communication monitoring technique on both the trunk and regional circuits have insured quality of facilities by early warning of troubles.
3. Preventative maintenance and redundancy of CPC and concentrator hardware have minimized down time.

Many enhancements have been, and are being, put into the NASDAQ System to supplement both broker and NASD supervisory functions. As a result, the maintenance programming group has been continuously involved in implementing changes, thus keeping their level of knowledge high, and allowing them to better handle problems in normal system operation as they arise.

Once a month, after system shut down, the Console Operators hold a refresher session, in which operating and recovery procedures are reviewed. During these sessions, hardware faults are injected into the system, to test the operators on recovery and reconfiguration procedures.

4. Fast response time has been achieved by:

a. Allocating high activity securities to a core resident dictionary and fast access drums. Lower activity securities go on slower drums and drum resident dictionaries. This allocation is an automatic file maintenance function.

b. Drum files are direct addressed, as opposed to normal indirect addressing.

c. Quote request calls access either a primary record or a secondary record, depending upon the respective queue lengths at the time for the data sources.

d. First frame responses are modified as required, by updates and stored on drum ready for transmission in final form. Frame #2 and beyond are compiled, as needed. Since all requests are for a minimum of frame #1, this procedure reduces processor occupancy.

e. Multi-processing and tasking allow up to seven (7) calls to be in some stage of processing at any one time.

f. Nested polling and priority reinvite schemes used with the regional multi-drop subscriber lines tend to equalize what would otherwise be an imbalance between transmit and receive line traffic.

g. Univac supported Exec modifications, along with the Availability Control Unit, provide 35 second completely automatic programmatic drum re-boot of the Exec and Application programs in the event of a program fault or loop.

h. Periodic file snapshots logged to tape, coupled with the on-line generated tape log of all calls coming into the system, provides for a second level of recovery.

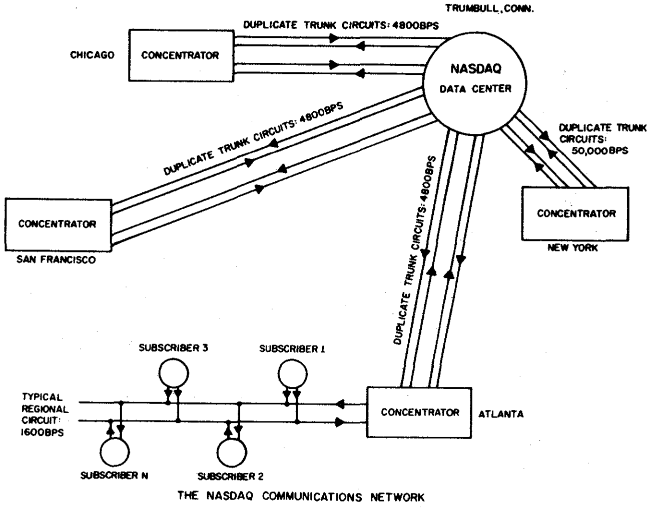

TRUMBULL, CONN.
DUPLICATE TRUNK CIRCUITS: 4800BPS
CHICAGO
CONCENTRATOR
NASDAQ
DATA CENTER
DUPLICATE TRUNK CIRCUITS: 4800BPS
DUPLICATE TRUNK CIRCUITS: 50,000BPS
CONCENTRATOR
NEW YORK
CONCENTRATOR
SAN FRANCISCO
DUPLICATE TRUNK CIRCUITS: 4800BPS
SUBSCRIBER 3
SUBSCRIBER 1
TYPICAL REGIONAL CIRCUIT: 1600BPS
CONCENTRATOR
ATLANTA
SUBSCRIBER N
SUBSCRIBER 2
THE NASDAQ COMMUNICATIONS NETWORK

Figure 7—NASDAQ block diagram

The NASDAQ System has been operative officially since February 8, 1971. The daily call rate has grown from approximately 300,000 calls per day at startup, to approximately 1.2 million calls per day currently.

The system is currently serving approximately 1,500 terminals in over 1,000 brokers offices. These are the Level 2 and Level 3 terminals in OTC Trading Offices. Indirectly, through Level 1 quotation distributors, such as Bunker Ramo, Scantlin, and Ultronics, approximately 40,000 additional on-line terminals are supplied with representative Bid/Ask quotations out of the NASDAQ System.

We are currently experiencing traffic rates of 75 to 100 calls per second during the peak of market activity.

The system now handles over 3,400 securities, including over 100 third market listed issues, from five (5) exchanges.

The system has just recently been opened up to serve non-NASD members with Level 2 service. This market area primarily covers the large individual and institutional investors.

With Phase I having established itself in the OTC community, interest is now turning to future possibilities for use of the system. Under discussion now with the NASD, are plans to incorporate into the system some phases of trade reporting, with tie in to clearing operations.

Tie-ins with regional exchanges on a pilot basis have already begun with the Philadelphia, Baltimore, Washington and National Exchanges.

The NASDAQ System, with its flexibility and nationwide communication complex, has stirred the imagination of the entire brokerage industry.

The visibility gap in the OTC marketplace has been bridged, and in doing so, the ground work has, we feel, been laid for future and even more encompassing changes and improvements in visibility and trading approaches for the entire brokerage industry.

The Weyerhaeuser information systems—A progress report

by J. P. FICHTEN

Weyerhaeuser Company
Tacoma, Washington

Just as every company is unique, so too is the requirement for information to operate the business effectively. For this reason, Management Information Systems cannot be directly transferred from one business to another. However, the experiences, conquests or "well dones", and pitfalls or errors of one company can serve as guidelines to another in providing Management Information Systems capability to their unique business.

In the 1950's, and early 1960's, Weyerhaeuser was aptly characterized as the "sleeping green giant". Business was good, but there were ominous clouds on the horizon. It was determined in 1963, by the board of directors, that Weyerhaeuser should be a growth company. At the same time, it was decided to guard against explosive unplanned growth. The key leverage points of healthy growth were identified as improved business planning, better utilization of financial strengths, and the development of management capabilities.

The objective to expand management capabilities led to programs for management education, formal planning processes, use of management science, and systems and data processing.

I'll concentrate today on the systems and data processing effort. In 1962, there was some EAM processing being done in accounting shops at the headquarters and some of our field locations. We had a data communications system which was transmitting punched paper tape from one location to another to speed up the processing of orders.

Outside of the EAM shops and the small exposure to communications capabilities, there was little familiarity on the part of management with systems or data processing concepts. On the other hand, there was a conviction on the part of the president that we had to get involved with computers if we were to be a growth company.

In 1963, George Weyerhaeuser, the head of the lumber division, initiated a study by a team composed of lumber division line management and outside consultants. The purpose of the study was to determine how systems and computers could be used to improve the lumber division's handling of business transactions and management capabilities. The study concluded that the existing information system was too slow, lacked the information storage and retrieval capabilities required, and was too costly. The study team recommended a computerized approach to handling orders and basic production and accounts data, and to provide all levels of division management with timely information necessary to make sound business decisions.

Specific recommendations were that they build an integrated MIS for the Wood Products business, utilizing an on-line integrated database for orders, inventory information, product information, and customer data.

As a result of the study, a systems director was established during the initial phases of detailed specification, and representatives of the three major business functions (manufacturing, marketing, and finance) were on the team. The team also included a technical director to solve the pure system problems of detail specifications for systems and hardware, file design, and programming. Later, when the computer was received, a computer center manager was added to the team.

This system was conceived as a communications-oriented data processing system with Wood Products locations scattered throughout the United States, communicating to the computer from Plan 137A teletype terminals and receiving information from the computer on a teletype output device.

The computers used were General Electric 235's and Datanet 30's. Special programming was required in both the communications processor and in the main frame to provide the capabilities required by the company. This was done with relatively low resource requirements in a relatively short time. I'd like to interject a comment here, that the high measure of success for the minimum amount of resources led us into a trap as we approached the next phase of our systems effort.

Once the Wood Products systems were running, the order processing time dropped from two weeks to two days or less. The visibility of our order position and the market improved, we were able to react promptly to rising or falling markets, and our shipping delinquency rate dropped. The success of the system and the large benefits received by Wood Products were well publicized within the company.

By 1966, managers in other phases of the business were heavily interested in getting a computer system like Wood Products for themselves so that they, too, could reap these benefits. At this time, a key decision was made to centralize the major computer facilities within the company, and to establish corporate control of the data processing functions within the company. Incentive for this was the capability of developing a common corporate database of information required to run the businesses. Cross functional files were also visualized as sources of information for management decisions in all facets of the business.

At this point, we went into Phase II in the development of Management Information Systems within the company. In 1966, there was no facility on the market to perform the functions required of a large computer system operating with a large common database. A decision was made to proceed with software development which would give us an operating system with the necessary capabilities, a highly sophisticated communications systems for processing both messages and data with restart and recovery capabilities as an integral part of the total configuration. The decision was made to proceed with the WEYCOS operating system as a joint development of the General Electric and Weyerhaeuser companies to provide these capabilities based on the success of the G235 project.

At that time, the operating system available from General Electric was GECOS-II. This was the nucleus upon which WEYCOS had to be built. WEYCOS was to have the features of input/output device independence, large database capacity, concurrent database usage, database recovery for aborted jobs and hardware failure, user transparent system restart, transaction processing via process codes, job scheduling and selection tailored to meet the needs of the business, passive message capabilities, and minimal operator intervention during the operation of the business day. WEYCOS was to operate in conjunction with a sophisticated communications network, which would be supported by communications processors with customized software.

Perhaps a brief explanation of what is included in WEYCOS, its outstanding features, and some of its pitfalls, might be appropriate.

WEYCOS is used as the exclusive operating system on one of our four GE-635 computers. The configuration of that system is as follows:

- 1—GE—635 processor
- 1—IOC
- 256K of memory
- 5—dsu 270 file electronics with 25 storage units
- 8—60KC tape drives
- 4—120KC tape drives
- 1—card reader
- 1—card punch
- 1—printer
- 2—Datanet 30's interfaced to the communication network.

The communication network consists of Western Union plan 137A dedicated circuits supporting the following:

- 31 circuits @ 10 C.P.S.
- 10 RO model 35 TTY's
- 81 ASR model 35 TTY's

WEYCOS is an operating system designed to control the execution of on-line and batch processing jobs on GE-600 line equipment.

The operating system provided by GE at the time WEYCOS development began in 1966 was capable of local batch operation only. Remote batch capability was in the process of being developed but was not available. Weyerhaeuser's data processing needs far exceeded the abilities of the standard operating system. So a joint development was undertaken to provide an operating system that would satisfy the needs of Weyerhaeuser Company and at the same time allow a computer manufacturer to gain experience in development of operating systems designed to utilize a large database.

The standard operating system provided by GE at that time was called GECOS-II; the jointly developed one is called WEYCOS which stands for **WEY**erhaeuser **C**omprehensive **O**perating **S**upervisor. This new system was an extension of the standard rather than a replacement. Under the WEYCOS concept, the main computer is just another station in a nationwide teleprocessing system. It has a station code and messages (JOBS) can be addressed to it just like any other station.

Some of the extended features include:

- A large central database (maximum size 16 million pages of storage with 1920 characters per page).

- Simplified job submission procedures—fewer control cards.
- Automatic restart and recovery of the database for protection against erratic user modification and/or system malfunction.
- Automatic job scheduling—at a certain time each day, etc.
- Automatic restart of programs making users unaware of system problems. (To effect absolute database integrity.)
- Data collection features—data, submitted as messages, is collected until needed for processing.
- COBOL and FORTRAN language extensions allowing users an easy method of reading and writing messages to and from the teleprocessing system.
- Monitoring capability greater than normally available to the computer operators via the console typewriters.
- Interface with a store and forward message switching system.
- Extended **I**ntegrated **D**ata **S**tore (IDS) processing capability to allow simultaneous access and update of database files.

These concepts, techniques, and operational methods must be thought of from a 1966 point of view. Some manufacturers provide some of these capabilities today; nobody did in 1966.

Characteristic of WEYCOS is the existence of a multiprogramming uniprocessor environment, a full complement of peripheral types, including a large database on disc, multiple programs in memory simultaneously, remote terminal capability, and elimination of external input/output queues through a centralized control structure.

WEYCOS is organized into several logical elements; they are:

- Control Structure
- Database Management
- Restart and Recovery
- Input Media Conversion
- Monitor
- Allocator
- Program Dispatcher
- Termination
- Output Media Conversion
- **I**ntegrated **D**ata **S**tore (IDS)

Each module, although functionally independent, interfaces with other modules. The more frequently used elements reside permanently in memory, while less frequently used sections are called from disc storage and temporarily used as needed.

As each message, in user terminology called a "JOB", is presented to the system from a Datanet or local card reader, its presence is recorded in the Control Structure for use in subsequent job selection. During execution, an activity interfaces with the Dispatch module of WEYCOS which, as necessary, accesses other parts of WEYCOS. After an activity is executed, it is processed by the Termination module for any necessary postprocessing. Peripherals and memory that were assigned to that activity are de-allocated, and the activity leaves the system, usually sending a response to the Datanet.

During activity execution, the system allows common access to database files with full user and hardware malfunction recovery. This is accomplished by the Database Management module interfacing with the Restart and Recovery module on behalf of the user program.

WEYCOS maintains internal control over jobs in execution and awaiting execution through a queuing and storage system called the control structure. The control structure is stored in memory and on disc. It contains dynamic information concerning jobs in the system, and static information describing system resources, resources required to run each job, response time, and business priority of each job.

The **C**ontrol **S**tructure **M**anager (CSM) performs several functions. When system resources are available and jobs have been queued preparatory to execution, CSM reviews the list of available jobs versus available resources to determine which jobs can be placed in execution. If the system is on schedule, CSM selects jobs for execution in the sequence of their requested start times. If the system is behind schedule, CSM selects by sequence of requested start time within business priority, starting with the highest business priority.

A job terminates in one of two ways—either by normal termination, if it completes execution without encountering errors, or by abort termination in the event that errors are detected during its execution. In either case, Recovery and Restart gains control, performs some processing, and then passes control to CSM. At this time, it can release the resources that have been utilized by the job during its execution and consider these for reassignment to other jobs not yet in execution.

The primary function of the **D**ata **B**ase **M**anager (DBM) is to coordinate access to database information. DBM performs all database operations in terms of physical units of information called pages. Each group of pages (i.e., file) is called a Page Set, and as such is controlled by a **P**age **S**et **M**anager (PSM). An example of a PSM is the **I**ntegrated **D**ata **S**tore (IDS) executive, which we will discuss later. Each PSM is responsible for the organization of information within its pages. All communication between DBM and the PSM's refers to page numbers within a Page Set.

Centralized control of the database has several advantages. DBM controls the distribution of pages across available hardware by means of a mapping function. This function consists of a number of algorithms used to convert page number within page set to a physical device address. Mapping algorithms can be easily changed; in fact, this was the major change required within WEYCOS in order to substitute DSU-270's for the previously used disc devices.

DBM also provides protection against database hardware malfunction. If requested by the PSM, it records the contents of each updated page on magnetic tape (after alteration image) as well as on mass storage. If any area of mass storage is unreadable or cannot be written on, DBM automatically invokes the appropriate recovery actions, so that the system user is unaware of the failure.

If a program in execution commits an error which is detectable by the hardware or the operating system, this program will be placed in a bypass status if it is a production job, or removed from the system with appropriate notification to its originator if it is a test job. In the former case, if the job has been referencing a production database, the contents of the database will be restored to its condition prior to the execution of the malfunctioning program at the time its failure is detected. If the failure of the program is thought to be due to an intermittent hardware failure, it may then be rerun at manual option. In any event, the database will have been protected from erroneous modification or multiple modification in the event of rerun.

The **R**ecovery and **R**estart (R&R) function within WEYCOS is intended to protect the system against detectable hardware and software malfunction. R&R will restore the database if a slave program which updated it aborts. This is the most common type of failure, and the R&R function is necessary to preserve the integrity of the database. This type of recovery does not affect the system as a whole, except that other database jobs are temporarily suspended while the recovery takes place.

Another function of R&R is to recover and restart the entire system, rather than a single slave job. Restart is responsible for recovering the control structure to a point known to be correct, reestablishing any remote traffic which was in process when the system came down; initiating slave database recovery, and rescheduling all slave jobs which were in process. Except for tape mounting, restart is done automatically.

The Input Media Conversion module reads the job from the input file that was preprocessed, generates control tables to be used by the Allocation module, and stores the job into its queues. The control cards are uniquely identified by a $ symbol in the first column and a system control word in columns 8 through 13. Through the interpretation of the control cards, Input Media Conversion creates files on disc for the many activities within the job, and channels pertinent information into tables for the use of the Allocation module.

When presented to the Input Media Conversion module, each job is considered to consist of one or more dependent activities. Allocation is based upon the individual activities of a job, and Input Media Conversion segregates these activities and presents them in an ordered manner to the Allocation module.

The Allocation module of WEYCOS performs the scheduling and allocation of peripheral devices and memory for each individual job activity. The scheduling algorithm of the Allocation module provides a practical foundation for initiating multiprogramming. The function of the Allocation module is governed by five guiding principles:

1. No peripheral device or storage is assigned to an activity until all the peripheral requirements for the activity can be satisfied.
2. No job activity is initiated until all preceding activities are completed.
3. Look-ahead through job activities for peripheral allocation.
4. An urgency criterion permits delayed activities to get first consideration each time allocation occurs.
5. The scheduling algorithm encourages an efficient blend of processing and input/output activities so that delays attributable to setup are minimized.

The Allocation module utilizes information files generated for each activity by the Input Media Conversion module. One file is the Allocator Table, which is a core table providing immediate reference information to the Allocation module. A second file is the Control Stack, which is a disc file containing complete peripheral requirements, operator instructions, core storage, and time limits, and the activity definition.

As each activity is considered, the Allocation module scans the Allocator Table to determine the gross peripheral requirements for that activity. If the requirements are met, specific peripherals are allocated and instructions are issued to the operator from information contained in the Control Stack. This technique allocates peripherals to activities in advance of execution, as determined by available peripherals, thus allowing operators to perform preparatory functions (such as mounting tape reels) while prior programs are in execution.

When the assigned peripherals are ready and sufficient memory is assigned to the activity, the Allocation module initializes the program and assigns the activity for in-process Monitor control.

The order of job allocation or scheduling is based upon a dynamic urgency concept. Initially, jobs are processed by the Allocation module as they are presented by the Input Media Conversion module; however, each time that a particular activity is scanned in the Allocator Table and its gross peripheral requirements cannot be met, the urgency of the job is incremented. When a threshold of urgency has been passed (that is, the urgency value has exceeded a predetermined limit), the entire job assumes maximum urgency and no other jobs receive consideration for allocation until the urgent job requirements are filled.

In summary, Allocation as described satisfies the first four principles listed at the beginning of our discussion on allocation. Regardless of the mix of processor-bound and input/output-bound programs to be processed, the Allocation module will attempt to assign components for complete utilization of the GE-635 system, thereby facilitating the achievement of effective multiprogramming.

The Monitor oversees the execution of each activity. Its functions include:

- User communication with WEYCOS.
- Processor-detected fault interpretation.
- Control of Master mode entry from job programs.
- Policing of the WEYCOS memory overlay area.

The user and operator interface with WEYCOS is performed under the Monitor module control. Operator/ WEYCOS communication via the input/output typewriter is a function of Monitor.

The Monitor interprets the various fault conditions that can be detected by the Processor. In cases where the fault is of the type that can be recovered by the user program, the Monitor module allows user-provided routines to be substituted for WEYCOS fault routines. These routines can be activated by the user through a fault vector.

Master mode entries are also translated by the Monitor. The result of an entry translation causes the Monitor to call upon other WEYCOS modules and routines.

The least used of the routines are not permanently stored in the WEYCOS reserved storage, but are contained in disc storage and, as needed, brought into an overlay area of the WEYCOS memory which is policed by the Monitor module.

Monitor also contains the system loader that is used to call system programs as required to process jobs presented to the system.

Individual activity and overall job termination is initiated by the Monitor module either at job program request or whenever it is determined that an activity must be removed because an error was detected. A portion of the Termination module is an overlay that is brought in from disc and placed into the memory area reserved for the use of the Termination module. Termination provides these functions:

- Provides a postmortem dump if the termination was requested via a forced termination condition.
- Communicates with the operator for the removal of files when needed.
- Closes the systems output file.
- Summarizes the output file information for Output Media Conversion.
- Produces an accounting record on the systems output file.
- De-allocates peripherals.
- Deletes appropriate Input/Output Supervisor, Dispatcher, and Allocator Table entries.
- Forces database activity completion (normal and abnormal).
- Deletes control structure reference to terminating job.

When these functions have been accomplished, a transfer of control to that portion of Termination that resides in permanent WEYCOS storage occurs, where core storage is compacted. This is made possible through the use of the Base Address Register which makes all programs dynamically relocatable. After completing these tasks, the Termination module relinquishes control to the Allocation module for reallocation of the released peripheral devices and memory.

The **I**ntegrated **D**ata **S**tore (IDS) is implemented as standard software for the GE-625/635. The form in which it is normally available, however, is not suitable to perform the multiuser/multiaccess type of operation required within WEYCOS. Hence, IDS has been extended so as to fulfill the requirements of WEYCOS users. The extensions made to IDS are:

- Multiple number of users are allowed to access and modify the data store concurrently.
- Special locking features are provided to prevent multiple users from making concurrent reference to the same data within the database. If not properly controlled, such references could cause incorrect results.

- A method of operation referred to as "test mode" is provided to simplify IDS program testing in this environment.

Modes of operation for IDS jobs in WEYCOS:

Q-Mode—Read from possibly changing file.
R-Mode—Read from unchanging file.
P-Mode—Protect against concurrent updates. (Allows multiple users to modify concurrently the same file.)
X-Mode—Exclusive access to the file.
T-Mode—Test mode changes are made to a scratch file. The accessed file remains unmodified.

This operating system was designed to provide a large corporate database capability. The needs of all operating divisions were to be met. Size was the first limiting factor. It was increased significantly. Multiple users of the database were also necessary. This is provided by special modes allowing users to specify if multiple users (at the same time) are allowed. Automatic restoration (of the database) to a known state in case of abnormal termination is also a must as files cannot be left in an unknown condition.

Remote teletypes used for inputting orders cannot be expected to follow the normal job submission procedures. These terminals are usually operated by clerks and use the same program many times a day. To facilitate this type of operation, the control information required by the system to start a job is stored within the system. It is accessed by a trigger called a process code and the data. The system uses the process code to tie to program records which contain pointers to program and job control stacks. Program records contain information on computer resources needed to execute this job. Station records describing every station known to the system are provided to simplify programming. Users address the desired station without worrying about whether it is a 33 or 35 teletype, a GE-115 computer, a GE-400 or GE-600 computer. The operating system prepares the output the way the specific terminal needs it. The user does not have to worry about folding lines, special slew characters, speeds, etc. This is very useful when special forms are used in certain teletypes for orders, shipping reports, inventory, etc.

Many business applications have a specific cutoff point every day. Shift changes are an example. WEYCOS has the capability to automatically start jobs so that they are finished or started at a certain time. The cutoff times are known to the system through process records which are user-supplied and permanently reside in the system. Along with this, end-of-day techniques are provided. There are two types: logical end-of-day from the operator's point of view, and business end-of-day from a user's point of view. Business end-of-day can be started whenever wanted, i.e., shift changes, office closure, etc. This is necessary to know where you are in round-the-clock operations.

Large computer systems are very complex interactions of hardware and software. This is further complicated by communication lines which aren't as precisely tuned as computer hardware. Errors, faults, and problems do occur. When troubles do occur, the programs in execution either stop or produce unpredictable results. Users cannot afford service interruptions. A large part of the WEYCOS system is concerned with automatic recovery and restart. This restart must be invisible to users. They want their jobs processed and do not even want to know about system problems, much less have to worry about them.

To accomplish this, snapshot pictures of the system are periodically taken and recorded on magnetic tape. They are taken at specific time intervals, i.e., three minutes or when significant events happen, i.e., certain on-line processing functions occur. Then, if the system fails, it can be rolled back to a known point and restarted.

Many business computer applications process data collected over a period of time. WEYCOS provides the capability to accumulate data for use later. All entities "flowing" through a WEYCOS network are called messages. These could be administrative from one teletype to another, jobs addressed to the computer, or just data. Jobs addressed to the computer are called active messages. These can be complete job stacks (all control cards included) called indirect jobs or the trigger-data type called direct jobs. Data only, addressed to the computer, is called a passive message. It is collected and journaled until called for by a processing program. This is especially useful in collecting data on equipment performance. Programs themselves can generate active and passive messages (i.e., spawn jobs). This makes an easy way to count the number of times a certain program is executed per time period.

To enable programmers to use the new features and capabilities of WEYCOS, COBOL language extensions had to be provided. These were primarily in the areas of database operation and message file accessing. Standard GE software includes a package to handle a randomly stored database. This package was modified to allow a larger size database, and concurrent users under special conditions.

A system such as WEYCOS requires more than the computer operators to monitor what is happening. Recognizing this, a control function has been provided.

A teletype station serves as the monitor. A capability called bypass is used to control the system load. All jobs entering the system must have a process code. A process code can be used by more than one user. If trouble has been reported with a certain program, the process code associated with it is put on bypass. Messages (jobs) trying to get at that program are put on bypass until the trouble is corrected. Then the process code and messages are removed from bypass and can become candidates for execution. This monitor is really the stethoscope into the system. Message queues are watched, output queues are counted, etc., helping to ensure that the system keeps operating. This monitor capability is also used to change program, process, and station records. Thus, the operating system does not have to be reassembled whenever changes are wanted in some critical tables.

The ability of the WEYCOS system to initiate the execution of programs on the advent of messages is unique among general-purpose operating systems. It is this ability which makes WEYCOS particularly well suited for the on-line execution of Weyerhaeuser's business transactions. It is not feasible to expect the vast majority of employees in large business to know or understand programming and the peculiar set of rules and procedures to submit programs to computers for execution according to conventional methods. The ability of WEYCOS to respond appropriately to messages representing normal business transactions in normal business-oriented format enables any normal intelligent clerical employee to utilize the system without undue or unusual training or effort in performing his normal tasks.

The WEYCOS system is capable of executing up to eight users' programs concurrently. This capability provides for a high system throughput which in turn should provide adequate facilities to handle the routine data processing and normal business transactions for a wide range of business activities. In addition, the system contains special recovery features which protect the database from being garbled due to malfunctioning programs and prevents production jobs from being lost in the event of malfunctioning hardware or software.

A computer controlled by the WEYCOS operating system is available to all users connected on Weyerhaeuser's leased line network. The computer is tied into the store-and-forward message switching network and had a unique station code. Programs developed by the central staff, in higher level languages (COBOL, FORTRAN), can be addressed by users from any station. Users can submit *jobs* to the computer to be run against previously written programs or against programs submitted with the job. Under WEYCOS, this is a batch process, not conversational or time-sharing.

With these thoughts in mind, it might be good to review the performance of the WEYCOS system.

- Peak processing rates of 893 messages (JOBS) per hour. These jobs are heavily IDS oriented running in a multi-access database with full recovery capabilities.
- Downtime of less than 1 percent. (both hardware and software).
- Average recovery time from a system fault is 12 minutes or less.
- Full automatic database recovery of 100 percent for all the aborted application jobs.
- Periods as high as 27 days with no software or hardware failures.

SUMMARY

WEYCOS is a success but also leaves us in a quandary. We have what we consider the most advanced and capable operating system available to us today. We are in a position of continually answering the question from our management, "Can't we get off of WEYCOS and go to standard software?" The answer, from the users of the system is always, "No". The users want and need the capabilities and comfort factors that WEYCOS provides.

We have looked for but not found a replacement. We know it is coming but can't say just when. We feel we are ahead of the field; we want to switch to vendor-supported software, but will have to wait till they catch up, although some vendors are getting close. In the meantime, we are caught in the vise between costs to maintain unique software and the availability of the capabilities the users require. For example, it is costing us about $140,000 a year to support WEYCOS, if you include memo billing charges for computer time.

So we come to the end of Phase III with some uppers and some downers. What does the future look like? Obviously we have learned our lesson the hard way and as a result we have had to learn to plan our future more thoroughly. We've established the system planner role as an integral part of the business planning process, and to serve as a communications link between the business and the systems organization.

We have established a Weyerhaeuser approach to uniform system implementation, which calls for review and approval at various steps through the development process by the business to insure that the business manager is getting what he thinks he is paying for.

The director of the Business Systems department has established minimum standard configurations for computers located remote from Tacoma to insure compatibility and interchangeability between locations. In addition, guidelines for the development of common transferable and modular systems have been developed so that each remote location servicing a specific segment of the business is not continually reinventing the wheel.

This is where we are in the '70's. What do we see coming in the future? We see small user-oriented smart terminals assuming an ever-increasing role within the system configuration of the company. We see more comprehensive database management than has been practiced before; database management that will reduce redundancy and insure completeness of the information contained, while maintaining database flexibility, processing efficiencies, and easy access. This leads to the ability for managers to easily query files for information.

This leads to a continued shift in skill requirements, not only within the systems organization but within the various businesses' organizations themselves. The computer will be more of a tool of management rather than a temple of knowledge.

We see more use of management science in systems with the concepts of projections, distributions, and statistical analysis becoming more and more a tool of management at all levels. We see a continued economic emphasis on systems in the areas of applications, development, specialized query languages, and operating systems themselves. We are projecting more concern for long-range planning, supported by a staff manual which will guide the planning function through its various intricacies and provide to management the information required to support the needs of the business.

We have the computing power in place to meet these requirements, such as full local batch processing capabilities, timesharing and remote batch from high-speed and low-speed terminals. Today, our load is shifting rapidly to batch processing. We see a future shift to interactive processing—a capability that only vendor-supported software will meet.

We expect that systems costs will require economic justification just like any other business investment. Our senior management has set this direction very clearly.

> "We believe that the involvement in the business and departments in planning, evaluating, and designing systems is critical and must be continued."
>
> "We want a continuing tough look at both ongoing and new systems."
>
> "We don't want to discourage new systems, but we do want them to face a tough justification test just like any other capital project."

We have learned many lessons that will provide practical building blocks to develop and maintain cost effective computer resources that will meet Weyerhaeuser Company's everchanging management demands.

The future of remote information processing systems

by M. J. TOBIAS and GRAYCE M. BOOTH

Honeywell Information Systems
Phoenix, Arizona

INTRODUCTION

Remote information processing is the computer community's wave of the future. For both computer users and for computer vendors it represents an enormous opportunity. To the user, it is a powerful tool with great potential to increase his business efficiency and profit. To the vendor, it opens a broad new market. Properly used, it can benefit both.

This paper discusses remote information processing networks in the rapidly growing area of on-line business applications. We have chosen to survey trends in this area, pointing out some of the important considerations which the potential designer or user of such a system cannot afford to overlook.

One aspect of remote processing is the use of multiple computer information networks. Large complex networks, such as ARPA, represent the leading edge of remote system technology. However, the authors feel that the majority of on-line business applications in the 70's will be implemented as individual systems, not as information utilities. Large scale use of shared or public information networks for on-line business applications will not be seen in the decade of the 70's, mostly because of problems of control, security, and privacy.

TRENDS

Much has been written recently about "the changing computer markets of the 1970's" and the industry's "coming of age," and still more in regard to the increasing sophistication of the computer user. Today's users are:

- Demanding system solutions to their business problems.
- Using an information systems approach to the control of their geographically dispersed organizations.
- Insisting that the emphasis be on INFORMATION as well as on computer power.

Basically, then, we can conclude that there is an increasing trend toward business oriented information systems whose characteristics are:

- Use of a data base—often large and possibly distributed among several branch offices.
- Communications orientation for information collection and distribution.
- Use of information processors (computers) for transformation and manipulation of information.

The needs which these users are expressing have of course been present all along. The historical objections to the computer have been:

- It is located far away from the field "where the action is."
- Only computer professionals can talk to it.
- It is very difficult to organize and store data within it.
- Somehow it always seems more suited for scientific than business problems.

Today these problems are gradually being solved, allowing the emergence of effective remote information processing systems for business use.

There are strong indications that the trend toward remote information processing is a significant one, and will continue to be so. Figure 1 shows projected shipments of new build computer equipment (for the industry as a whole) for batch vs on-line use. While the market estimates are relatively firm only through 1975, the trend can be projected with reasonable assurance farther into the future.

Though various market surveys may differ in detail,

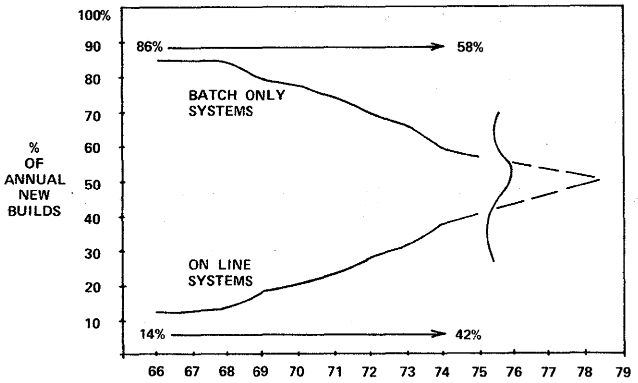


Figure 1—New build shipments forecast

all seem to agree that on-line information systems will be of increasing importance in the decade of the 70's.

APPLICATIONS

Before looking into the future of remote information processing, it may be interesting to briefly survey applications of the past and present. This can indeed be done briefly, since remote information processing is still in an early stage of development. This of course is one of the reasons that its future fascinates us.

Yesterday's applications

Remote processing began in the 1950's. The earliest examples were special purpose systems, and were largely concentrated in a few types of applications.

Airline reservations

In 1952, American Airlines installed the first computerized reservation system, which employed a special purpose computer, Teleregister's Magnetronic Reservisor. Several other airlines installed identical or similar systems. The first break in this pattern occurred in 1958, when Eastern Airlines installed a Univac File Computer, Model 1, marking the first use of a general purpose digital computer for airline reservations.[1]

Command and control

One cannot speak of early remote systems without mentioning SAGE. This on-line command and control system for the U.S. Air Force was planned early in the 1950's, and became operational in 1958.[2]

There were many other pioneering remote information systems. However, in general the cost and complexity of on-line systems deterred their use except when there was no alternative. The airlines, for example, simply could not handle their increasing reservation loads except through automation. Similarly, SAGE represented the only possible means of detecting and reacting to an enemy attack employing missiles and jet aircraft. The absolute need for these systems therefore overrode cost considerations.

From our current vantage point in time, we can see that other applications had a similar serious need for on-line processing, to provide improved efficiency, cost reductions, and competitive advantages. However, these potential advantages were not recognized at the time, so these (to us) very real needs were ignored.

Today's applications

We shall look at today's remote information processing applications more generally than we looked at yesterday's. The accompanying papers in this session describe some interesting examples of today's uses. We will therefore speak generally of the types of applications which are installed, not specifically of the companies or agencies which use them.

On-line banking

Many of the larger banks have already moved into the use of on-line teller terminals to provide instant balance checking and updating.

Order entry

These systems, with the necessary supporting inventory management features, are in fairly wide use today. They are usually applied in the area of wholesale, rather than retail, sales.

Retail sales

Credit checks are being increasingly automated. A number of retail establishments have installed terminals at each cash register, so that credit can be checked prior to every sale.

Point-of-sale automation is being carried farther, in some cases, to encompass not only credit checks but sales slip preparation and even simultaneous inventory updating. One of the pioneering efforts of this type, the TRADAR system jointly developed by General Electric and the J.C. Penney Co., was discontinued, for reasons apparently not directly related to its technical feasibility.[3]

Computer aided instruction

CAI is being used in some schools, but is not yet in widespread use. Student teaching methods using computer-controlled consoles range from quite straightforward drilling to sophisticated interactive tutorials.

Law enforcement

On a small scale, law enforcement systems have many of the same requirements as the armed forces' command and control systems. Government funding is being heavily applied to these systems, especially the FBI's National Crime Information Center network, and the local/regional Law Enforcement Assistance Administration-funded systems.[4]

Network utility

It would be unrealistic to complete any discussion of today's applications without mentioning the ARPA network. While beyond the scope of the more common applications, it has amply proved its feasibility. It is now operational, tying together over 25 computers (as of early 1972) in a research-oriented network. The ARPA design goals are built around providing ½ second access from a terminal anywhere on the net to a host anywhere on the net, regardless of the number of nodes separating them. The great variety of host computers involved—Digital Equipment, Honeywell, IBM, Burroughs, etc.—and the consequent enormous potential for resource sharing make ARPA unique.[5]

Tomorrow's applications

It is now time to gaze into our crystal ball at the applications of the future. In general, we see these as logical outgrowths of today's systems, expanded in scale and scope, but not dramatically different in basic concepts. However, this growth and broadening of scope will enormously increase the computer's influence on society, as on-line systems move more and more into the mainstream of the average person's life.

Retail sales

There will be nearly universal computerization of point-of-sale recording, including instantaneous credit checks. The average retail store or supermarket of the late 1970's will have its cash registers on-line. They may be locally on-line, to the store's private computer, or linked to a regional or national network of many such stores.

This trend will also extend into a more hostile environment, the automobile service station. On-line credit checking, and possibly on-line account updating, will be standard.

Medical systems

On-line hospital management will be widely used. The computer will also be used for the storage of medical histories, and to aid in diagnosis.

Increased population mobility makes centralized medical records almost a necessity, and this need will be met via remote information processing. The ever-increasing pace of developments in medicine makes it impossible for any doctor to remain abreast of current trends, even in his area of specialization. An information processing system, however, can be kept up-to-date, so that all current and/or obscure diagnostic information is available to every doctor.

Law enforcement

Increasing crime rates will provide the incentive for further use of computers. The criminal, like the population in general, is increasingly mobile, forcing improved intercommunication between law enforcement agencies. The local/regional networks now being developed will gradually merge, creating a national network tying together all levels of law enforcement agencies.

Financial utility

Banks will continue to expand their use of remote information processing. On-line banking may merge (although perhaps not within this decade) with other systems to form the often discussed "financial utility."

The financial utility will be the means of creating the "checkless society," or even the "moneyless society." Each citizen will have a "money card," which he will use for all purchases. When the card is inserted in an appropriate terminal, the buyer's credit will be checked and, if good, his bank account will be reduced and the store's account increased.

This application will require a network of computers similar to that in ARPA. Certain technical problems (mostly of scale) are involved; however, these may well not be the limiting factor. Human reaction to this form of money management may be sufficiently negative to delay or limit its acceptance. In some form, and at some time, however, it is bound to come.

Computer utility

For our final example of tomorrow's remote information processing applications, the computer utility seems the obvious choice. Much has been written and spoken about the computer utility, and it is difficult or impossible to evaluate many of the predictions.

In speaking of the computer utility, we mean the very general, highly reliable, information network which provides data, message, and computing services to an extremely wide spectrum of society. We also mean systems utilizing large and flexible information processors to provide low cost computing to the average person—the home owner, the housewife, the student, and so on—as well as to business and industry.

In many respects the computer utility represents the "goal" of remote information processing. When the day arrives on which the computer utility is both feasible and practical, all of the problems discussed during this session will have been solved. The most important of these problems will relate to the acceptance of the utility by a society confident of the reliability and data integrity and security of the total system. When (or possibly if) this day arrives, the computer will indeed have had an ineradicable effect on society.[6]

BUILDING AN INFORMATION NETWORK

The implementor of any remote information processing system must realize that he is faced with a networking problem. Every remotely oriented system requires a network, where a network is defined as a set of hardware and software elements which provides for the collection, processing, and distribution of information.

The problem of creating an information network with optimum characteristics is logically the same regardless of scale. A small information processor surrounded by a few terminals forms a network. So does a nationwide hookup of half a dozen geographically separated information processors, tied together via communications lines, interfacing with hundreds of terminals, as shown in Figure 2. The major difference is in scale, not in the techniques of networking.

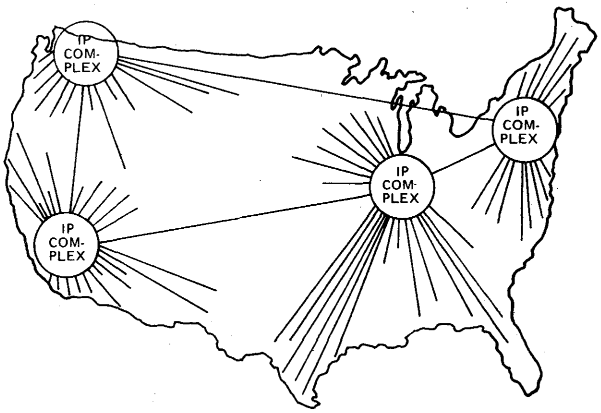


Figure 2—Large information network

Before discussing, in the following sections, the hardware and software elements needed to form an information network, it is important to point out the characteristics shared by all remote information processing systems.

Flexibility

As a company becomes increasingly involved with remote information processing, the business becomes closely intertwined with the system. Thus, the information system must be as flexible and capable of growth as the business itself.

Availability and integrity

A company which implements a remote information processing system (for purposes other than simple time sharing) is committing an important part of its communication and control structure to that system. When a company commits its business (or part of it) to the information processing system, its dependence on that system becomes very great. High availability, or continued operation of the system, is therefore required. The user must carefully evaluate the degree of availability needed; this is an expensive commodity, but an absolutely essential one.

Provisions for protecting data integrity are also essential. The heart of any remote information processing system is its data base. These large data bases are often the only repository of vital business data. It is not only impractical but impossible to maintain this information in any manual form. It is therefore absolutely essential that this data be protected against loss or damage.

Optimum cost/performance ratio

Remote information processing systems are usually more expensive than batch systems, but many are more costly than necessary. There are many techniques for reducing costs, especially communications facilities costs, while retaining desired performance. Some of the commonly used cost reduction techniques, such as multiplexing, concentration, and line switching, are described in the following section on Network Hardware Elements. These must be thoroughly investigated by the user who wishes to install an effective remote information processing system.

NETWORK HARDWARE ELEMENTS

An information network is formed of a number of hardware elements, which can be classified as follows:

- Terminals
- Distribution facilities—lines, trunks
- Network (communications) processors
- Information processors
- Multiplexors and/or concentrators
- Switching devices

Some of the more important features of each of these elements are discussed in the following sections.

Terminals

The proper choice of terminals is important to the success of any information network. Great flexibility in terminal design is important, since the terminal is the part of the system closest to the user and with the most need to adjust to his individual requirements.

Many categories of terminals will be used in the 1970's, including teletypewriters, CRT displays, remote batch, data preparation, and graphics. The two most important types, however, will be displays and application specialized terminals.

Displays will normally operate on voice grade lines, at line speeds up to 9600 or 10800 bps (the higher speeds will be needed to satisfactorily perform screen filling/emptying operations). Displays will most often be used on multi-drop lines, in poll and select mode, with occasional use on single station private or dial lines when heavy utilization makes multi-drop lines impractical.

Application specialized terminals will include teller sets for banking, factory data collection devices, stock broker sets, airline reservation sets, and other similar devices.

These specialized terminals will be heavily used in transaction processing systems. The terminal operator will, in many cases, use the terminal only part-time, and will therefore not be an expert operator. His lack of specialization as a terminal operator will often be compensated for by terminal specialization.

Distribution facilities

The distribution network is the means by which information flows between nodes in the network. Distribution facilities are provided by the common carriers—AT&T, Western Union, etc.—or potentially by the newer carriers such as MCI and Datran.

Digital networks will represent a significant advancement in the time frame beyond 1975. Both AT&T and the specialized carriers are beginning to install such facilities. The advantages of such networks include:

- Emphasis on digital traffic
- Improved switched line facility
- Low error rate
- High availability
- Significantly reduced cost

The reduction of user cost is a primary advantage of such networks. In the Datran network, for example, regional transmission (i.e., transmission beyond the local trunk) will be independent of distance. The charges proposed are much lower than current rates, especially at the higher transmission speeds of 9600 or 14400 bps.

It is concluded, then, that distribution facilities are becoming more flexible, more reliable, and more economical. This trend should continue as competition strengthens in this field. The resulting improvements will be manyfold for remote information processing.

Network processor

A network processor is a computer specifically designed for the control of data communications. Network processors are vital to the success of remote information processing systems. A network processor can be used either as a front-end to an information processor, or as a free-standing system performing functions such as complex concentration and/or message switching.

The important reasons for the use of a front-end network processor are:

- Off-loading of network oriented functions from the information processor.
- Greater flexibility and versatility.
- Increased system availability.

It has been found in practice that the network processor design should emphasize flexibility and memory capacity, rather than the maximum in processing speed. Its primary tasks are those of data manipulation and transliteration, not "number crunching." However, the instruction set of the network processor must necessarily lend itself to efficient character and byte data handling. If this is not the case, processing speed may indeed become a limiting factor. It has also been found advantageous for the front-end network processor's instruction set to be closely related to that of the associated information processor.

Some of the important design requirements of a network processor are these:

- Memory size up to 256K bytes.
- Fast access disk interface (the disk space in front-end applications should ideally be shared with the information processor in the "delta" configura-shown in Figure 3).
- Optional console, card reader, and printer interfaces.
- Optional non-interruptable power source.
- Ability to be configured with full redundancy.
- Program controlled reconfiguration and communications line switching.
- Interfaces to narrow band, voice band, and broadband lines, both dedicated and dial-up.
- Ability to support several hundred concurrent user messages—possibly as high as 1000 in 1975 and beyond.
- Floating dial-out channels.
- Integrated modems.
- Ability to interface with other network elements such as multiplexors, line switches, etc.
- Ability to interface with more than one information processor.

The above listed requirements are probably not fully met by any network processor design in existence today. Also, the rather large load requirement listed is representative of only a very few present day applications. However, the trend toward large loads and very high availability requirements is unmistakable. Thus, these requirements are not blue-sky—they are for tomorrow's applications.

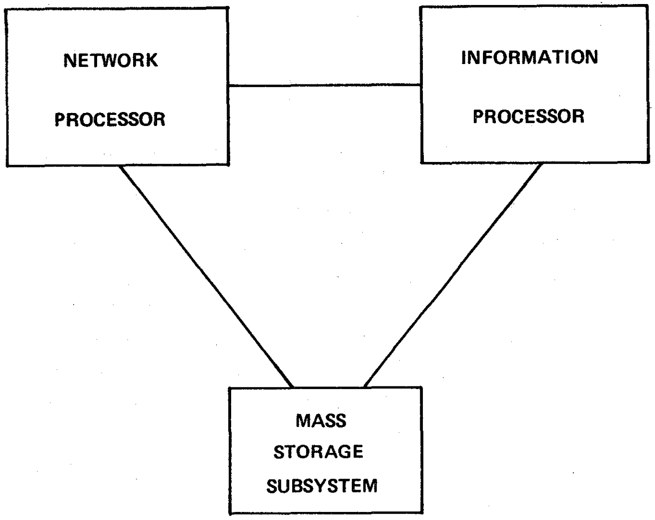


Figure 3—"Delta" configuration

Two aspects of the network processor requirements are sufficiently interesting to warrant further discussion.

The "small" communications oriented user requires the same degree of sophistication and functionality as the large user. Thus, the growth of the front-end network processor function on site is largely related to load handling capability. This makes the network processor design even more difficult, in order to provide for the "small" load economically yet grow to the very large loads as well.

Though the communications front-end processor has, in the past, been thought of as merely an instrument of the information processor, it is now clear that it must function quite independently of the information processor. For example, the front-end processor must remain operational even when its information processor fails. In this way contact with the rest of the network is not lost, input messages are still received, output messages can still be sent, and—most important—full service can be brought back with the minimum of difficulty and time delay after information processor repair is effected. Front-end processor independence is also needed as this processor takes on more and more tasks related not to information processor interface, but to the network (i.e., message switching).

Information processor

The information processor is of course where the actual application processing takes place. Perhaps the most important hardware feature of an information processor to be used for remote processing is modularity.

In a remote system, the information processor should be capable of at least fail soft, and for some applications fail safe, operation. Fail soft is aided by hardware modularity, with the possibility of configuring multiple processors, multiple memory modules, multiple I/O connections, and so on. This modularity must be such that failure of any hardware element, such as a memory module, does not prevent continued operation of the remainder of the system.

Multiplexors/concentrators

Both multiplexors and concentrators can be used to greatly decrease communications line costs. Multiplexors are passive, non-programmable devices, while concentrators are programmable.

A multiplexor splits up a single communications line so that it can carry a number of multiplexed messages concurrently. There are two types of multiplexors, Time Division (TDM) and Frequency Division (FDM).

A wide range of manufacturers offer TDM's and FDM's. General Electric, Timeplex, Computer Transmission Corp. (Multitran TDM), and GTE Information Systems are only a few of these. AT&T also offers multiplexors.

A concentrator is a form of network processor, whose function is to compress data on many lower speed lines onto fewer higher speed lines. Because it is programmable, a concentrator is more flexible and potentially more powerful than a multiplexor. As with other network processors, it is also capable of assuming other network oriented tasks.

The Honeywell Model 730/50 is a typical concentrator built around a minicomputer. It concentrates up to 128 low speed lines (up to 300 baud) or up to 64 medium speed (up to 2400 baud) onto 1 to 4 medium speed lines (up. to 10,800 baud) leading to a remote network processor or information processor.

Switching devices

Reconfiguration line switches are used in many networks. Their most common use is to switch communications lines between two or more network processors.

Figure 4 shows a typical fail safe system. The reconfiguration switch shown might be the Honeywell Line Transfer Device 355, which can switch up to 96 lines under either manual or program control. The diagram also shows a "Deadman Timer," such as the Honeywell Computer Monitor Adapter, typically used for automatic failure detection in configurations of this type.

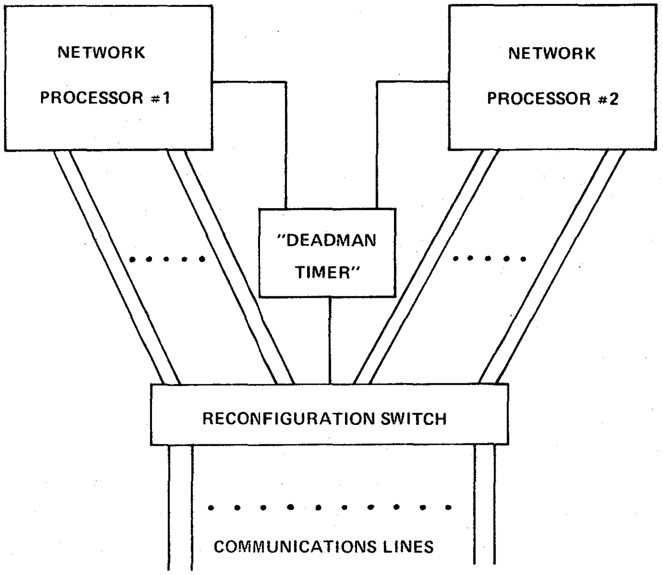


Figure 4—Fail safe configuration

SUPPORTING NETWORK SOFTWARE

The implementor of an information network must also look closely at software requirements. It is not possible, in a paper of this length, to discuss—or even list—all of the functions needed. Therefore, only a few of the more important are included here.

Terminal interface

The system interface seen by the terminal operator must be appropriate to the application, and to the operator's level of training. Particular care must be taken to support the type of operator who uses the terminal, not as a full time task, but as a subsidiary part of his job.

For these cases, the language used by the terminal operator must be simple and self-explanatory. Clear diagnostics must be provided in case of error, and software assistance must be available when the operator is presented with unusual conditions.

Load leveling

Some networks of multiple information processors may require a load leveling function. This allows jobs to be distributed between the processors to even out work loads. If the processors have unequal capacity, or are of different types, load leveling can also detect jobs too large to be run on a smaller system, or jobs which require a specific type of processor, and direct them to the appropriate location for execution. As large distributed information systems become more widely used, load leveling in various forms will be increasingly necessary. However, vendor-supplied software for this function is rare today (if not non-existent).

Distributed data base

As networks with distributed computing capabilities become prevalent, distributed data bases will naturally develop. A distributed data base exists when two or more information processors which communicate via the same network allow creation and sharing of permanent files.

Without going into the complexities of distributed data bases—which are many—it can be pointed out that a single job may require access to more than one part of a distributed data base. This can be achieved either by transmitting file data or by transmitting jobs or tasks. Both of these methods cause increased communications load, and may require very high speed transmission.

Message management

Applications which process remote input and output have, in general, a more complex task than those which handle local I/O. Ideally, an application program should be able to treat a message from a CRT identically with one from a card reader, and be able to send output identically whether to a CRT or to an on-line printer. In practice, it is seldom this simple.

One approach to this problem has been provided in the COBOL communications-oriented verbs RECEIVE and SEND. The approach is an unfortunate one, in the authors' opinions, since it would be more consistent to use READ and WRITE. However, CODASYL has chosen to use different verbs, so the COBOL community must accept this.[7]

One pitfall which the user must watch for in a vendor's implementation of these verbs is whether or not device independence is provided. CODASYL has not specified whether terminal characteristics are to be handled by system software or by the user program. The former, except in very rare cases, is much preferable.

For example, the verb RECEIVE should cause a line (or other logical increment) of input to be presented to the program, stripped of terminal control characters and blank filled if necessary (for short lines or imbedded horizontal tabs). If this is not done, a great deal of tedious—and sometimes complex—processing is required in the application program. Output should similarly be prepared as a line or page by the program, with all necessary terminal control provided by the system.

Although it makes no difference to the application where (outside of its boundaries) terminal-specific processing takes place, this is important to the system as a whole. Terminal characteristics should be handled entirely in the network processor, allowing the information processor to ignore these characteristics. This allows much greater flexibility to change/add terminal types, and/or to redirect messages to alternate destinations.

In the context of remote input/output, a software generator approach to creating the coding to handle terminal specifics is an absolute necessity. It is unrealistic to expect any vendor to provide terminal handling software for every terminal on the market. Instead, the use of a generator package can make it relatively simple to add types of terminals to the network. This is the only realistic way to approach the variability inherent in remote terminals.

Data base management

The designer of a remote information processing system must look very closely at data base processing, since this will be the heart of the system. A data base management software system of considerable sophistication is normally required.

The most important point to be considered, which is sometimes overlooked, is the need for concurrent access to the data base. If any of the on-line applications update the data base, there is a potential problem of access conflicts.

Even with update access, these problems can be minimized if each application can be structured to read only one record and hold it the minimum possible time before updating and releasing it. In this comparatively simple case a software system for record locking can prevent any other access to a record being updated.

More complex situations occur if one or more applications must update multiple records. Locking records in this case can cause the "deadly embrace" situation shown in Figure 5. The data base management software should detect and resolve these conflicts.

To improve throughput, locking may be used only for updating programs, with inquiries allowed to run without check. This approach can cause the inquiry mode application to see inconsistent data base status, as shown in Figure 6. Programs must be written to take this into account; otherwise they may malfunction when a "shifting data base" condition arises.

There is a considerable difference in complexity between the simplest case of concurrent update (one record read and updated by each program) and the most complex (many concurrent applications each updating many records).

Whenever possible, on-line applications should be structured to achieve the simpler case. One way to do this is to perform the minimum updating necessary on-line. For example, an order entry system might be set up to update credit limits, inventory balance, and production and shipment schedules. Analysis might show that update of the production and shipment schedules could be postponed for batch handling, either once daily (usually at night) or in slack on-line periods. This would significantly decrease the amount of on-line updating, therefore decreasing the overhead involved in locking records and monitoring for interference.

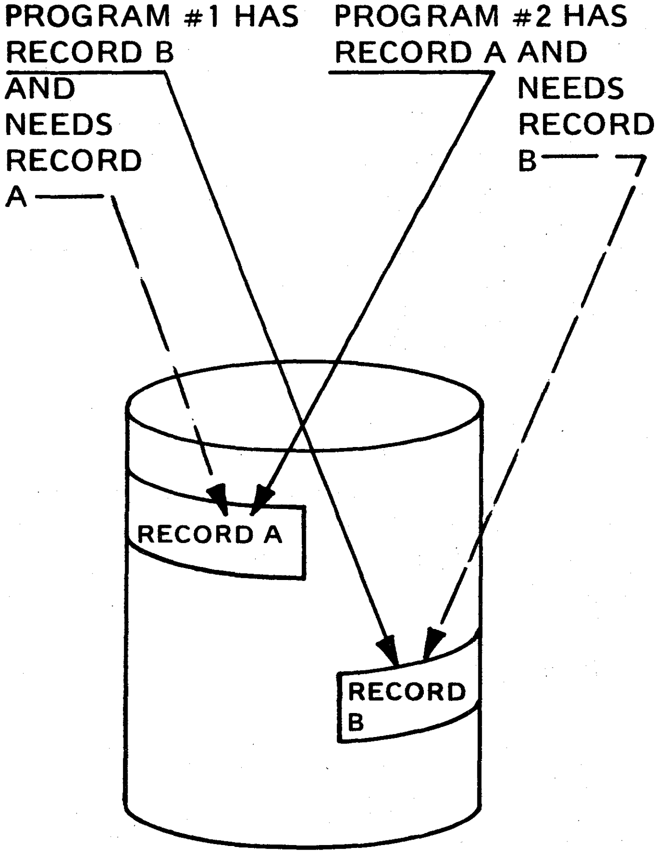

PROGRAM #1 HAS RECORD B AND NEEDS RECORD A
PROGRAM #2 HAS RECORD A AND NEEDS RECORD B
RECORD A
RECORD B

Figure 5—"Deadly embrace"

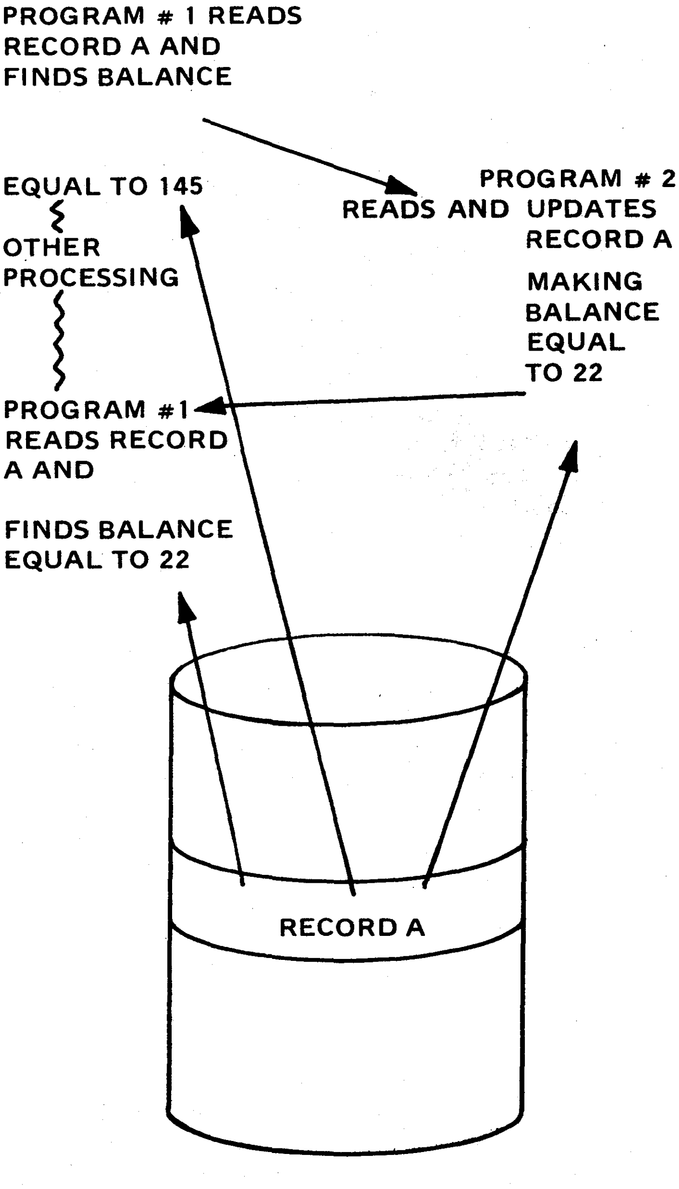

PROGRAM # 1 READS RECORD A AND FINDS BALANCE
EQUAL TO 145
OTHER PROCESSING
PROGRAM # 2 READS AND UPDATES RECORD A
MAKING BALANCE EQUAL TO 22
PROGRAM #1 READS RECORD A AND
FINDS BALANCE EQUAL TO 22
RECORD A

Figure 6—Shifting data base

This area should be extensively studied during system design. Unnecessarily complicated on-line update applications can be very costly in terms of software complexity, debugging difficulties, overhead, and in operational delays associated with locking and resolving access conflicts.

Integrity protection

Integrity is a broad heading, covering a number of hardware and software features. Both hardware and software must be properly oriented toward achieving adequate integrity.

Fail soft or fail safe configurations must provide graceful degradation when a non-essential component fails. Restart/recovery features must be provided, in case of total system failure.

Data protection is extremely important, so that data base information and incoming/outgoing messages will not be lost, duplicated, or garbled. Features such as file change journaling and automatic recovery are especially important, to protect against file damage in case of application program or system failure.

Test and diagnostic routines must be available, to test communications components, network and information processor and their components while the remainder of the system is processing normally. Off-line T&D is also necessary; however use of off-line T&D will generally be minimal, as most error diagnosis will be required while the system is on-line.

Security

Security (including privacy protection) features must include, as a minimum, terminal and user validation procedures independent of the information processor (because of terminal/terminal exchanges in some systems). Detailed user validation, including password and permission checks, may also be performed in the information processor. More complex security features, such as data encoding, may be required in some networks.

Debug aids

It is important that adequate debugging aids be available and specifically applicable to the on-line data base and business processing environment. Too often, on-line software systems are designed for production use only, forgetting that both initially and on a continuing basis system/program changes will be required. The prudent user will avoid this trap by insisting on adequate test features.

In testing on-line applications, it is very helpful to be able to easily load and test an on-line application from any convenient location, perhaps a teletypewriter, perhaps a card reader. To allow this, the software system must provide device independent I/O, and also the ability to obtain snapshots and traces interactively during test execution. Interactive debugging, when properly supported, can enormously improve programmer productivity.

Testing must often take place concurrently with production work. This is particularly true in systems which process actively for all or most of the 24 hour day. In these systems, all debug aids must be structured to allow concurrent test and production operations.

Finally, aids are needed to allow testing of data base update programs. The best aids allow test programs to be run, simulating update, without actually altering any file data. If test aids are not available, the user will find that he must manually set up test data bases, and quite likely also alter his programs so that they can use these data bases.

CONCLUSION

Remote information processing systems are technically feasible today. The most complex forms of such systems, information networks, are being implemented in small—but growing—numbers.[7] The ARPA network referred to earlier is an outstanding example of these.

These networks represent the leading edge of technology in at least some respects. They have proven the feasibility of distributed information systems, but not necessarily the general applicability of this approach. Most of the remote information processing of the 70's will still be in the form of individual on-line business systems. In many cases these will be networks which include multiple information processors, but single user owned. The mass use of public networks to solve business problems on-line will not be seen for some time, largely for reasons of security, privacy, and control.

Much work remains to be done to improve software technology, particularly in the areas of message management, data base management, and integrity protection. Innovative users, perhaps more than the hardware/software vendors, are leading the attack on these problems. The papers which follow describe some of the systems which embody today's state of the art in remote information processing systems.

ACKNOWLEDGMENT

The authors would like to acknowledge the contribution of Hal B. Becker, from whose work on structuring and organizing Information Networking we have borrowed freely.

REFERENCES

1 R A MC AVOY
Reservations communications utilizing a general purpose digital computer
Proceedings of the Eastern Joint Computer Conference
The Institute of Radio Engineers Inc NY NY 1957

2 R R EVERETT C A ZRAKET
H D BENINGTON

SAGE—A data-processing system for air defense
Proceedings of the Eastern Joint Computer Conference
The Institute of Radio Engineers Inc NY NY 1957

3 R M PETERSEN
TRADAR: Death of a retailer's dream
Datamation Vol 17 No 11 June 1 1971

4 P HIRSCH
LEAA: Who guards the guardians?
Datamation Vol 17 No 12 June 15 1971

5 L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
AFIPS Conference Proceedings SJCC May 1970

6 D F PARKHILL
The challenge of the computer utility
Addison-Wesley Publishing Company Reading Mass 1966

7 *First report of the Communications Task Group to the CODASYL programming language committee on the COBOL extensions to handle communications processing*
COBOL PLC Item No 68114 May 1969

8 D J FARBER
Networks: An introduction
Datamation Vol 18 No 4 April 1972

Interactive processing—A user's experience

by HERBERT F. CRONIN

The First National Bank of Boston
Boston, Massachusetts

INTRODUCTION

During the years 1965 to 1968, The First National Bank of Boston developed an active interest in the concept of cost savings that could be obtained from major accounting operations through the use of interactive processing. The amount of clerical effort which could be saved through the installation of an effective real time system was believed to be quite large, and, if so, the reduction in the labor costs could support a rather large computer charge and still generate significant net savings. The operation which appeared to be the best candidate for a pilot project was that of the Bank's Factoring Division, an enthusiastic computer user for ten years. In fact, the desire on the part of the Division's Management to obtain a real time system considerably pre-dated the availability of off-the-shelf hardware and software that could be obtained within reasonable limits of expense and risk. The Factoring Division is primarily responsible for an accounts receivable operation involving one-hundred and forty thousand accounts and a total of some two million invoices which are posted to these accounts each year. When these invoices come due, the customers mail their payment checks to the Bank, and the Bank then matches the checks to the specific account and invoice or invoices which are to be paid. The invoices are then paid and removed from the file and appropriate printed reports are prepared. At any given point in time, there will be about four hundred thousand invoices and related items open on the file.

The accounts have headers of fixed size to which are attached a variable number of invoices. The size of an account varies from about four hundred characters on an inactive account to about one-hundred and fifty thousand characters for several of the very largest and most active customers. The total account file includes about one-hundred and fifty million bytes.

Prior to the introduction of the real-time system, there was a staff of about three-hundred and twenty clerks who processed the accounts receivable work. This complement was adequate to handle the average daily transaction volume of about twenty-five thousand entries, but periods of overtime were required at seasonal business peaks. The skills of the staff were varied, ranging from that of keypunching from completely prepared source documents, to that of bookkeeping, a function which involves considerable responsibility and judgment in the handling of cash and the allowance of deductions.

This clerical organization had been designed to work with a batch processing computer system. There was little interchangeability of function among the staff groups because each of the groups was structured around a file of computer printed documents used for that one function. Each morning, the new ledgers and other reports were received from the computer and distributed among the clerks who then filed the sheets into their individual buckets. As a result of the document filing requirements, it was difficult for the Division management to reallocate its staff from function to function as the workload varied.

EXPECTED OPERATIONAL BENEFITS

The primary benefit which the Bank hoped to gain from a real-time system was one of a reduction in the unit cost of work processed. "Instant" information had no monetary value in this case, but the centralization of all data onto a single file would be a great value. The major advantages of the real-time system which could reduce the labor content of the operation were expected to be:

- Availability of the data base to anyone in the division who had a need for information. This facility alone would eliminate the need to print and file bookkeeper's ledger sheets. It would also almost completely eliminate the input effort and delay associated with special request ledger sheets, which were used by other sections as the main vehicle for obtaining information from the file.

- The ability to validate transactions during the original input.
  The batch system was always subject to a reject rate of about three percent, and these "kickouts" created a need for a large and complex manual control system to insure that they were researched and re-input without being lost.
- The potential for performing a complete set of daily proofs on the computer.
  It had not been possible to implement these proofs in the batch system since the items rejected from the system caused all of the larger accounting categories to be out of balance. Therefore, a manual balancing system was required. In the new system, all transactions would be kept in the file.
- The installation of an account locator program which could help the staff identify the accounts to which they wished to post transactions.
  The batch system required the use of the computer account number on each item input. The process of determining this account number was clumsy, and it often required the use of special files, both computer printed and manually recorded, which were some distance from the work stations of the people needing the information. The account locator was designed to allow users to input transactions by account number, by a simple name code which could be derived from the name and address of the check received, by the DUNS number, or by a bank checking account number taken from the MICR encoding on the check.

The general expectation was that the Factoring Division could improve the operation and reduce the staff by making the data base available to those who needed it, by purifying the input at the point of entry into the system so that the file would be accurate and complete, and by eliminating the huge amounts of printing and clerical effort required to keep the old batch system in balance.

These concepts had existed at the Bank for a period of several years but no active steps had been taken toward starting a detailed examination of the feasibility of the project. During 1968 the Bank prepared a long range plan to replace its existing main line computers (which were then seven years old) with new hardware. This process had been started with the acquisition of a new computer for several small applications, and a master plan was developed which would cover the period during which all of the major application systems of the old computers were to be reprogrammed and upgraded onto new hardware.

PROJECT INITIATION

The master plan included the task of reprogramming the existing Factoring system, and it provided the opportunity to examine whether a complete re-design might be economically justified. Therefore, the Bank retained an experienced consultant to make a preliminary review of the potential savings, costs, and practicality of a real-time system. The evaluation was made in the Spring of 1968, and the consultant's report endorsed the real-time concept and estimated that hardware and software costs would be low enough to make the investment a profitable one. A detailed feasibility study was then initiated. Four systems analysts from the Bank, four from the consultant, and four from the Factoring Division were allocated for a period of four months to study the existing operation, and to write a formal feasibility report covering cost, systems design, and hardware specifications.

The report was distributed in September of 1968. It included an overall system plan (a design which has proved to be remarkably accurate) an estimate of the number of programmers and the schedules needed, and a forecast of the computer hardware that would be needed. The cost model indicated that the recovery of the investment could be expected to take two and one-half years. Much of the actual design and estimation was done by the three senior systems people who were most closely associated with the project, one representative of the Factoring Division, one consultant, and one member of the Bank's Systems Research Division.

The report favoring the real-time system was submitted to senior management officers for their consideration and approval. No decision was reached at the first review meeting because the division heads involved in the project were concerned that the study team had not adequately examined the various other types of systems that could be designed. In particular, they requested that we study:

- The cost of this proposal in relation to the other alternatives (i.e., a new batch system), and
- The risk in undertaking a development venture of this type in view of the problems that other real-time systems users were reputed to have had.

In response to these concerns, the team spent an additional month in evaluating the costs for various alternative systems. This review found that the costs of reprogramming for a batch system would be less than those for the on-line system, but there would be no prospect for a reduction in expenses since the new system would essentially duplicate the old one. The only difference would be the new machine on which the system would operate.

The question of risk favored the batch system, which obviously had only a minimum of developmental unknowns. However, the team recommended that the real-time system be the one implemented, primarily because it felt that prospects of gaining a net profit from such a system would more than justify the risks assumed. Management concurred, and the decision to begin the new system was issued in November of 1968.

DESIGN OBJECTIVES

The goals by which the results were to be measured were:

- A saving of twenty-five percent of the operations staff.
- A computer running time of twelve hours per day, eight on-line, and four for the nightly runs.
- A response time of fifteen seconds per message (this response time appeared to be intolerably slow once actual experience was obtained).
- A time frame of twenty-one calendar months and planned level of four hundred and nine man months of labor.

START-UP

The start of any large project (this one was expected to have a staff peaking at thirty people) is largely consumed by the task of organization, personnel selection, acquisition of quarters and all of the other administrative details. This phase took one and one-half of the twenty-one months which we had forecast. Following this was the preparation of a formal, written project development plan which covered schedules, individual responsibilities, reporting, organization charts, and standards. It took another six weeks to complete this plan, and we realized that three months of our allotted twenty-one had gone by, and we had not yet begun a concentrated study of the technical alternatives. It now seems obvious that we did not properly plan for the time and manpower to handle the administrative details and the organizational work which proved to be so time consuming throughout the first half of the project.

DESIGN DECISIONS

The early months of the project were the period in which the major project decisions were reached. Seven of these decisions will be covered in the following sections of this paper, and, during the discussion, the reader may find it helpful to refer to the brief tabulation of system statistics to be found at the end of the paper.

Modeling

The first major decision that the project team reached was one in favor of building a system model. The level of effort that the development of this model would require was unclear, but the general belief was that it would be no more than several months before the model could be put into operation. The most pressing argument for the model was the feeeling that the batch processing experience of the team members could not provide an adequate framework within which real-time systems decisions could be reached. The first model used the IBM TPAD analytical procedure, and the results, even before the model was complete, convinced the team that the equipment on site at the Bank could not handle the CPU load that the system would produce. New hardware was ordered and the project costs were adjusted accordingly.

The individual models of the application programs were very crude during the initial runs of the TPAD system, and the subsequent development of the detailed specifications for the programs produced a constant stream of changes to the model, all of which seemed to be in the direction of the consumption of more and more of the available computer resources. At this point, the analysis indicated that the system would develop impossible queues, even with faster hardware, at a message level less than that needed to complete a day's work, and thus a new and more accurate simulation model was undertaken using CSS. This model was intensively developed over a period of about four months, and system modifications were made and simulated until the results from the model indicated an improved but still borderline set of response times. There was a considerable amount of disagreement within the Bank's technical staff as to whether the system would actually operate within the fifteen second promised response time for sustained periods. Many more changes were made in the file accessing plans, and new applications programs (such as those in the MIS area) were postponed, in order to reduce the CPU load. The use of the simulation was the key to the technical changes that were made during this period. All changes were modeled and those that produced improvements in the file accessing and CPU loads were incorporated into the plans. The importance of the simulation and its effect on specific programming decisions cannot be overemphasized. The lack of real-time experience on the part of our people forced them to rely upon the simulation for many decisions that might have been made intuitively by a more experienced team.

The final result of this phase of the modeling effort was a prediction that the modified system could handle the anticipated workload, but that there would be little room for expansion. The manufacturer restricted the availability of the CSS language at this time and the modeling efforts were stopped for an extended period of time. The current model used by the Bank is a GPSS model which has been developed as support for the new projects that have been started in the area of real-time processing.

The use of a drum

The primary characteristic of this system was the reading, repetitive on-line updating, and writing of large accounts into and out of the disk file. One account could be searched, displayed, and updated many times in order to complete one transaction. This file handling problem created a need for an intermediate speed storage device and to avoid the extensive delays that would occur if all I/O were done from the disk. The early simulations confirmed our assumption that adequate response times could not be reached without some storage device which was considerably faster than a disk, the fast access to data stored with a block of large core storage appeared attractive but the slow net transfer rate made it ineffective for very large units of data such as those processed by the Factoring System. The drum, while slower in accessing due to latency time was actually much faster because it could transfer records at channel speeds. A drum was chosen and it has proven to be an effective and reliable device.

Real-time update

The question of updating the file as transactions were processed or writing the transactions onto a log tape for later processing, was decided by the savings to be gained in the Factoring Division's operations. The system could only produce the expected personnel savings if the payment process were to eliminate items as soon as the payment was made. The real-time update, which had been the original plan, was again confirmed by the project team. This decision, however, created the potential for future trouble. Normal problems such as program errors, power failures, and the file problems could not be cured by a simple re-run as in the past batch systems. The initial availability of the system to the user suffered as a result of this, and adequate control over the problems was obtained only after an extended period of system operation had led us to develop safeguards against the common types of file errors.

The decision to update on-line implied that the CPU consumption of the system would be much higher than with a simpler data collection system. Because of this, several types of transactions were eventually switched to a batch type update in the nightly runs in order to reduce the daytime CPU load.

Pre-processor

The question of obtaining a pre-processor to reduce the load on the main computer appeared several times during the early stages of the project. The final decision against a pre-processor was an important one, since it was expected to set the policy for future Bank systems.

The benefits of a pre-processor were expected to be:

- A reduction of about 30K in the CPU core storage requirements.
- A reduction in CPU consumption of about 15 percent for the planned computer (according to the model).

The drawbacks were:

- The increased cost for the pre-processor, which would only be used for the eight hours per day of on-line time.
- The additional potential for vendor interface problems.
- The need to have a second pre-processor to provide backup.
- The need to learn to program a new machine and to provide all of the associated assemblers, libraries, and backup procedures attendant to the use of another manufacturer's software.

The cost of the two pre-processors appeared to be about $280,000, a very large price for a 30K and 15 percent CPU saving. Our multivendor problems (terminals—telephone—computer) were already troublesome enough and there was a distinct desire to avoid additional complications of this type. The decision was made against the pre-processor, and the teleprocessing load was assumed by the main CPU. There have been no reasons to doubt that, for this one system, this was the best approach. Since that time, the Bank has upgraded its computers twice in order to provide additional capacity for the other new systems being installed. The resultant increase in speed and core means that the share of the load attributable to the teleprocessing programs has decreased, for the computers have grown larger while the teleprocessing load has remained fixed. The impact of this load is now small enough to be of little interest.

Shared hardware

Cost allocation was the major factor in the decision between a smaller computer dedicated to the real-time Factoring system and a larger computer shared with other applications. The daily time requirement for the Factoring system was eight hours on-line and four hours for a reorganization and report preparation run at night. Any cost for equipment to be used only by this system would have to be charged in total to the division using the system, but any equipment which could be used by others would be on a shared cost basis. With only a half day's expected use, it was apparent that we should accept compromises in order to make the hardware usable by other applications and thus cut our costs. This overall philosophy has never proven to be a problem. At the present time, the concept has been expanded so that the teleprocessing and queue management programs can handle multiple applications concurrently. The hardware on which the Factoring system is now running will be processing three independent, large scale real-time systems by the fall of 1972.

COBOL

The next major consideration that deserves discussion is the use of COBOL for the on-line applications programs. The original opinion of the project team was that the teleprocessing and work queue managers should be written in assembly language in order to make them re-entrant and avoid the fatal problem of reloading the modules before each execution. The systems programs were, in fact, written in assembly language, but the Bank management was anxious to see the application programs written in COBOL. The method chosen to achieve this was to divide each application program into steps, each of which corresponded to the processing of one of the messages which formed part of a transaction. Each of these steps was given a step number within the COBOL application program. The programmer's working storage (which includes the step number) was required to be placed in a terminal work area (called TWA) which could be kept on the drum between uses. There was one terminal work area for each terminal. COBOL statements such as ALTER were prohibited, and the programmers quickly learned to keep their working storage confined to the TWA. Therefore, each time an application program was turned on, it found the TWA set as it had been at the end of the last step, already read into core and available for the next step as determined by the step code. The modules could then be serially re-used although only one terminal could be using a given application module at one time.

In practice, the steps tend to be short, and the programs are thus close to being re-entrant in the way the programmers use them. This feature facilitated the use of COBOL, and the CPU and core inefficiencies have proven to be tolerable.

CRT display consoles

The CRT versus typewriter decision was the easiest to make. There were no technical reasons why the CRT displays would be more difficult to work with than typewriters. The need on the part of the user was for the display of rather large quantities of data for which no printed record was needed. We therefore selected CRT displays and obtained them from a vendor other than the CPU vendor. No programming problems were encountered and the CRTs operated as expected.

PROJECT RESULTS

The results of the project have been most satisfactory and the system performance relative to the original goals is as follows:

- The clerical savings began to develop as soon as the new programs went into production, and, once the period of installation problems was over, the full savings were achieved. Our recent experience has convinced the Factoring Division that some additional savings, which were not in the original plan, can be obtained through the development of new programs.

- The computer running time is eight hours on-line and three hours at night for a total of eleven hours. The productivity of the CRT consoles in this system is so satisfactory that the Factoring Division is investigating the possibility of cutting the on-line hours per day or of reducing the number of terminals used to service the operation.
- The response times are in the three to six second range and only a few exceed the original (but excessively long) fifteen second response time estimate.
- The development time was about double the estimate, but during much of that extra time the project consisted of a five-man maintenance team which was finishing the periodic programs and special reports. This is the one category in which the project failed to meet its goals, and we will be more cautious in the future in estimating the time consumed in the learning of a new technology. If the project were to be started today, with the experience we have already acquired, the estimate would be twenty-four months.

The present satisfactory state of events was, however, preceded by a long period of agonizing problems, frustrating delays, and excessive costs. The installation of the on-line programs created a constant series of failures which resulted in frequent re-starts and the need for the users to re-input data. The telephone lines used to communicate between the computer center and the user's quarters were a daily source of trouble, and the only solution which worked was to move most of the users to the computer center and communicate by means of a direct cable within the building. Two telephone lines remained in use after the move, and these eventually became completely reliable.

Once the communications line problems were solved, the Factoring Division hit a seasonal operations peak and the system developed impossibly long response times. The CPU and I/O consumption problem which had occupied the team's attention during the modeling effort had not actually been solved, and CPU utilization rose to a level of 90–95 percent with long queues being developed. The evidence gained from a comparison of the actual system against the model indicated that the I/O in the model was based upon "good" transactions and efficient programming. Neither of these was a realistic assumption, and the rate of messages input for each transaction processed was about double that which had been expected, due to batch proofs, wrong account identification, re-tries, and all of the usual human errors. The programs, as well, were discovered to be performing unnecessary I/O. Although the systems and the clerical problems were somewhat relieved by program changes, the final answer was found in the replacement of the computer by a faster one which fortunately was already scheduled in accordance with the Bank's long term plan. Response time has averaged five seconds since that time, and the users feel that improvements beyond that point cannot yield increased clerical efficiency.

Once the response time problem was eliminated, the production rose and we encountered running times for the nightly reorganization and report preparation run of up to ten hours. Times of this magnitude meant that a normal re-run due to a program error would often delay completion of the nightly runs until well into the working hours of the next morning, thus preventing the system from going on-line at the normal starting time. Our availability suffered as a result, and some backlogs of work appeared in the user operations. A concentrated effort in improving the inefficient parts of the job has cut the running time to three hours per night. The availability of the on-line system gradually increased during the same period to its present level of 97 to 98 percent. (Note: The availability is calculated by dividing the sum of the hours each terminal was actually available by the sum of the total possible terminal hours.)

CONCLUSIONS

Reflecting on the course of events during the project and upon the final results that were obtained, there are several important lessons which we have learned (or re-learned) that may be of value to others who are considering real-time systems projects for large clerical operations areas.

Project potential

This system was installed for purely financial reasons. The original estimates of a 25 percent personnel reduction through the elimination of paper handling and the double processing of some work turned out to be easily attained, and the computer costs have dropped below the planned level because the system shares the cost of a larger, faster machine with other departments. The net annual savings, therefore, are higher than anticipated. The cost overruns for the project have extended the payout period, but there is evidence to suggest that the lessons learned will have the effect of improving future estimates and of reducing future real-time system development costs.

Risk

The experience that the Bank's technical staff had gained in its twelve years of programming large computers was assumed to have a carry-over into the area of real-time systems. All were conscious, however, of the risk of entering a new technology with such a large project. The early months of the development went quite smoothly, but when the system began to come on-line there was a traumatic and extended period of trouble with such unfamiliar items as modems, telephone lines, and CRT controllers. Furthermore, the traditional problems of program error, power failure, or machine trouble now could cause highly visible system crashes and leave bad records in the data base. New disciplines had to be learned in order to bring these problems under control. It often seemed that the failures, especially in the area of remote communications, were as new to our vendor's representatives as they were to the Bank, and that both were learning about the equipment at the same time.

The fire-fighting efforts required to keep the system running took more and more of the development team's time, and the rate of progress toward completion of the project slowed to a halt on several occasions. The scheduled completion date was passed and management began to become very concerned with the availability problems and with the project slippage, especially since these had been recognized as potential trouble areas from the very start. At times, the problems were so numerous that the achievability of the project goals was in question.

Gradually the problems diminished, then disappeared, and the remaining programs were completed. An overhaul of several key programs was undertaken, and the revised programs saved large amounts of time and CPU usage. The Bank's subsequent interactive systems projects have benefited from the Factoring system, and their risk is now no greater than it would have been for a batch system.

Management must be made aware of the potential for delays and interim problems in a pilot project, and the project team should realize that the ability to obtain satisfactory results from the on-line terminals during the early testing stages of the installation must not be extrapolated into a belief that all will be well under a full production load. It may be prudent, in some cases, to program all of the nightly runs and printed reports prior to allowing the team's (and management's) attention to be drawn into a demonstration of the on-line part of the system, for people may conclude that the project is nearly done when, in fact, only the on-line terminal programs have been finished.

Modeling

The use of a system model seems to be absolutely necessary in order to evaluate alternatives and to predict system capacity and performance. The user starting his first project in this area will benefit most of all, for the model will act as the substitute for past experience with similar systems. As the user acquires his experience, the modeling process may determine fewer major decisions, but it will provide a means of checking the decisions and predicting the performance improvements from faster computers or peripheral devices

File accesses

File accesses can be a major source of CPU problems in a large system. The file handling will often be provided to the application programs by means of macros, and careless use of the access macros can result in unneeded I/O as well as wasteful CPU Execution. In the Factoring system, the CPU requirement for computation and logic within the application programs is about one tenth of the CPU consumed by the file handling part of the system. In the Factoring system the files are large and the proportion of system load due to file I/O is perhaps larger than in other applications in other companies. Nevertheless, systems designers should count file accesses with great care in any real-time system, and regard every additional access to the file as a significant systems load. Each of the heavily used applications programs should be subjected to a review by several programmers to provide assurance that the accessing plan is as efficient as possible.

In summary, the system has met our expectations but the problems encountered during the development were severe and prolonged enough to place success in doubt. The experience gained has been invaluable, and the Bank is capitalizing on that knowledge by building two additional systems of a similar size. The writer believes that there is a great savings potential for systems of this type in other business areas and that their development and use should not be assumed to be limited to large firms.

STATISTICS—The First National Bank of Boston Factoring On-Line System

File Size	150 Million Bytes	Average Printing Requirements	200,000 lines
Average Messages per hour	in = 4,800, out = 4,800	CPU	IBM 370/155, 1 megabyte of core
Peak Messages per hour	in = 7,200, out = 7,200		
Average Message Size	in = 100 characters, out = 400 characters	CPU Utilization (On-Line)	35-40% in June, 30% (planned) in December '72
Teleprocessing	Remote, 1 second poll	Drum	2303 in June, 2305 in December '72
Lines	10 lines, 2,400 bps speed		
On-Line System Core Requirement	420K, 610K including OS	Drum Utilization (On-Line)	50-60% in June, 30% (planned) in December '72
On-Line System Running Time	8:45 a.m. to 4:45 p.m.	Disk	4-2314 Spindles in June, 2-3330 spindles in December '72
Response Time	82% under 6 seconds, .5% over 50 seconds		
Nightly Run Core Requirement	6 programs at 120K, 1 program at 600K	Disk Utilization (On-Line)	20% in June, 10% (planned) in December '72
Nightly Run Time	3 hours average, 4 peak	Terminals	58 Sanders Model 720 CRT Consoles

The myth is dead—Long live the myth

by E. L. GLASER and F. WAY, III

Case Western Reserve University
Cleveland, Ohio

Although the field of computation has been of the most explosively expansive in the history of mankind, still we find certain basic ideas which were valid many years ago are still assumed to be valid today. On the other hand, some apparent truths that were discovered at the beginning of the computer era unfortunately have been lost in antiquity and must be rediscovered, not once but many times. In this paper we do not expect to be definitive but at least to examine some of these myths, the true and the false, and perhaps to find some that should be destroyed and to uncover others to prevent younger workers from having to rediscover what each of us has had to do several times ourselves. To compound our problem we have the fact that the electronic calculator of the programmable variety is now starting to encroach and has already well overlapped the capabilities of the minicomputer. "This new machine is a hybrid. It overlaps both the minicomputer and the programmable desk calculator. The standard machine containing 1000 locations is capable of containing a program of over 2000 steps. In addition, specialized tape units using modern cassettes are available. The basic machine comes equipped with a number sufficient to handle all of your temporary storage needs, will enable you to bring in new programming systems, and it will even have sufficient facilities to permit you to sort blocks of data. As a consequence this machine will meet the needs of office, laboratory, or accounting room! Its size is moderate and will fit on a reasonably large desk. Cost is under $18,000. As a special added feature, its basic instruction set is implemented by using the most modern techniques of microprogramming and, therefore, may be modified in the future as the manufacturer develops new classes of instructions that will be of use to the various customers of this excellent machine. Programming systems that take full advantage of the human oriented decimal arithmetic are already available with it and, furthermore, all registers can handle both numeric and alphanumeric quantities! For further information circle . . ."

Such an ad does not seem in any way unusual. It could have appeared in any one of the trade magazines such as "Datamation" or "Computer World" any time within the last year. It appears to be an interesting architectural structure, and a rather interesting compromise between a desk calculator and a general purpose computer. Obviously this is a made-up example. Further, it should be obvious to the reader that this is meant to lead you down some kind of a primrose path. For those who have not guessed or for those who are not old enough to have remembered, the machine described has a venerable history and it is known as Univac I. The only thing that we have done is to rescale it in terms of modern technology.

The purpose of the preceding bit of fairy story was strictly to bring out the fact that our concept of what is a small computer and what is a large computer, what is a mini and what is a desk calculator have undergone radical changes, not just in the last twenty years, but in the last five. For those that don't believe it they may go through the exercise that we have done. A more interesting one might be to examine some of the new minicomputer offerings that take advantage of electronic MOS memory. At our university we have recently installed such a machine that also has built-in floating operations and memory protection hardware. This particular machine is being used as a minicomputer to support a special laboratory. It happens to be satellited to a much larger machine. Imagine our shock and surprise when, at the time the programming system was being established, we found it has the throughput capability of what everybody in this audience was calling a large machine as recently as 1965. We have purposely not identified either the minicomputer or the machine we compared it to. We leave that as an exercise for the reader and the hearer since after making this rather shocking discovery we looked at other potential pairs in this little game and found that it almost doesn't make any difference which ones you look at.

The purpose of all this has been to ask the reader to re-examine some of the "truisms" that have been around. The problem with any unwritten law is that you don't know where to go to erase it. Within the field of computer systems we are in an unusual position of being still a very young field with unbelievable number of customs, old wives tales, etc., that bind us and many of us are not even aware of it. This particular example we feel is a good one since it starts to challenge the most common of all such myths. "There is economy in scale." This is true. There is. It is also true that as our understanding of computing grows our aspirations for more and more computing also grow. However, the above example raises the question of do we really need as much economy of scale, for example, for interactive time-sharing where the main purpose is simple programming and editing. Even in the case where we are developing programs for very large and demanding problems that do require the modern equivalent of the largest computers, the question can be asked "is it really necessary that humans interact to develop these programs with the largest machines or could we not by means of modern technology place a simpler machine in the hands of the user for this purpose and subsequently transmit the problem when debugged to the large machine if the large machine requires it?" Not that long ago most people would have felt that being able to afford a Univac 1107 or an IBM 7094 for each person that needed computing was an unbelievably optimistic dream. The question is, now that we can do it, is that the way we are going to go?

SOME SACRED COWS FOR THE SLAUGHTER

The preceding sacred cow which was exposed is not the only one. In fact, it is the most commonly referred to myth of our field. At the time we were originally preparing this paper we identified four additional such canards that had to be examined and either accepted or rejected on their worth but not on faith. We find, fortunately, that at least one of these has been partially rejected. It applies primarily to the user of small machines. For years it was felt that it was possible to cut corners on small machines. One can have good debugging features and all kinds of aids for the programs in the large machine but since the small machine is so small one can cut corners and not give the user all of this "help". We are glad to say that this feeling no longer is rampant in the industry as it once was. Large machines can support large staffs. Small computers, minis, and programmable electronic calculators cannot support staffs at all and therefore must supply the help for the user. We regret to say, however, that the remaining sacred cows are still alive, healthy, and doing ecologically un-nice things to the programmatic landscape. The first of these is "well, I know the machine does have some problems in its architecture but we'll fix that with software". How many times have we heard that, either in those words or words similar to it? We would even suggest that perhaps some new terms be offered, that henceforth the software be called hardware, and the hardware should be called easyware. All of us know of systems where this particular concept was applied and the pieces were never picked up totally. There are even two or three machines that we know of, made incidentally by different manufacturers, where the pieces never were picked up. No manufacturer is immune from this particular set of comments.

The second of our three mythical animals for the slaughter is "of course computers operate in binary. That's how God meant them to, that's why he gave you two thumbs!" to paraphrase Tom Lehrer, who once pointed out that counting in octal was just like counting in decimal if you didn't have any thumbs. We might add that counting in binary is just like counting in decimal if you are all thumbs. We have always assumed in recent years that the machine must operate in binary, and we can store other representations such as alphanumeric and decimal in it if we so need because they can always be displayed on small indicator lights for the serviceman and after all the software will make sure it gets printed out on the printer. What can we say? Machines are finished and operating systems are finished but computing systems are never done. The programmable calculator people have learned this and are now delivering machines that are ready to use as soon as you uncrate them. Unfortunately, however, they are having to relearn what the computer industry learned some 15 years ago. They are having to deal with the problems of specialized functions versus generality and in some cases they are having to relearn the whole technique of algebraic scanning and interpretation. However, they have learned many of their lessons well. For those of you who don't believe it, go look at some of the modern programmable desk calculators that operate from an algebraic language. The fact remains that people still think in decimal despite the fact that many programmers pride themselves in being able to read in octal almost as well. This is not to say that binary manipulation is not required inside the machine. The question however, must be raised and re-examined as to whether it is better to compute in decimal when decimal answers are required. For those who think that this is a dead issue, when was the last time you tried to explain to a good hardnosed accountant why it was by putting his problem on a modern machine you could only come out with inexact answers?

The third sacred cow that should be examined carefully is that of secondary storage. The cache memory has come into quite wide acceptance over the last several years. Yet, most machine designers insist that secondary storage disk memories and drums, are really input-output. We might humbly suggest that the core or MOS memory is really a cache for the larger online store. The implications of this are broad, however, they should be examined carefully since ultimately if you want online storage, you've got to be able to get at it in some reasonable way and an integrated approach rather than the massive peripheral attack that has been mounted upon the problem of memory, memory organization, and name space, just might be fruitful.

One of our most widely regarded cows has apparently been strangled by a twisted pair of wires—the cow was "closed shop is better" and the demise is being caused by remote terminals running in either batch or interactive mode. It is indeed marvelous to view the influence of state of the art technology (a pair of wires) making such a pronounced change in the management of computation centers. Another beastie which is long overdue for extinction is the practice of letting circuit designers dictate the arithmetic properties of a machine. This has produced such anomalies as a machine which does not have a floating point multiplicative identity—try and explain to an irate user just why one times X is not X. and yet another marvel wherein raising a real number to an integer power is more accurately done by exponentiation and logarithms than by successive machine multiplication. To add further fat to the sacrificial fires we now have shirt pocket calculators with nine or ten DECIMAL digit floating point (+99 to −99) numbers, trig functions and inverses, etc. It clearly is high time for some specialists in numerical mathematics (note—not numerical analysts) to have a say in things.

As a concluding blast in the large scale animal division, let us merely note some atrocious characteristics of some of the higher level languages such as letting machine "features" to percolate thru to the point where the user *must* know about them, or languages which have made the compiler writer's job simple at the expense of the target user—again try to explain just why the user should not write

```
DO 403 X = −9.35,0.45,0.33
```

Of course we can get around the problem, but some users seem to think that the machine should help them solve problems rather than the other way around.

SOME SACRED CALVES

In this section we would like to put forth some sacred calves that have not had the chance to grow up yet. It is our fervent hope that they too someday may also be cluttering up the technological landscape and a latter day programmatic Don Quixote may sally forth to do battle with them. Our first candidate is already becoming well accepted. Don't put it in software if you know it isn't going to change. Put it in the hardware, it's easier to debug. If it is in the software, it's there because it needs to be parameterized and will change, either within one user's environment or between users, and change rather drastically. A parenthetical comment might well be inserted here that often is put in software for a somewhat different reason, namely "we don't really understand it so we'll give it to the programmers."

A second small critter has to do with microprogramming. Microprogramming really isn't an answer, it's a question. The question is, what is the place of interpreters? There is nothing basic about a microprogramming system. It is just that the old concept of interpreting in real time has been rediscovered. What is its future? Will we start seeing languages again that are interpreted rather than compiled and executed? Obviously we already have. A very well-known one is APL. Excellent work is going on in several centers including Harvard on languages and language systems in which both interpretation and compilation can be exploited quite interchangeably and continuously.

Our third candidate for immortality as bovinus mythicalus is in many ways the most important. A number of networks have sprung up in recent years. The work currently going on between a number of centers on the ARPA network may well be one of the most important information handling experiments that have been conducted in the last decade. Obviously it is of interest to demonstrate what can be done with long range, broadband communication that ties together a number of computing centers for load-sharing, for gaining access from one center to many other centers, and even more importantly, for making available computer resources at a distance so that each center need not have their own machine, but buys the service as they do "electric power", to cite a common cliche. The importance of this experiment may well transcend any of these reasons. As speed of devices increases the problems of the interconnection increase also, but at this point they are increasing at a much higher rate. We are rapidly approaching the time when it will no longer be possible to increase the speedy machine merely by increasing the speed of its elements. For those of you who are interested in circuit and hardware design, pray consider the dilemma if we were to offer you an unlimited supply of absolutely free components that were capable of operating in the one to two pico second range. These components, however, are sized about the same as our present IC dual-inline packages. What difference would they make? The answer is not much! We are already at the very edge of what present day packaging can do with present day organizations. Still, if we can network machines together across the country and have them work on some kind of a cooperative task, then obviously we can do it across a cabinet or even a PC board. This is not quite the same as array processors such as ILLIAC IV, rather it is looking to the time of learning how to use cooperative independent processes for the solution of large problems.

CONCLUSIONS

It is hard to say what will happen if we either follow the new dictums or keep the old. Prognostication in this field has been woefully poor. The majority of it has either been overly pessimistic or just totally off the beam. There is one conclusion that can be drawn. Whether these new dictums are the ones to be adopted or not, the fact remains that computer systems design is becoming much more complex than it ever was before. Systems design must do quite a bit more than merely putting software on existing hardware. What is needed is an integrated set of design tools aimed at solving users problems and meeting users needs. Where possible, the users needs should be anticipated since user behavior will change with the advent of new tools and the availability of new techniques. It is our firm belief that it is this specific problem of being able to handle the complexity of modern systems that has caused the rather noticeable slowdown in the change and design of large computers today and it is the lack of this inhibition that has promoted the burgeoning minicomputer industry. Computers have been designed as tools to handle large and complex problems. They have been applied by many different industries, not to just the implementation of technological solutions but to the investigation of complex technological design problems. Unless and until the computer industry adapts its own tools and finds workable mathematical models that will permit the handling of the inordinate complexity of modern day computing systems, the stagnation that has appeared in the medium to large scale computer area will continue and these medium and large scale computers could ultimately be swept away by new, small, high-powered minicomputers aimed at individual use or for the use of a few people together with cooperative networking techniques. It is our fervent hope that this does not happen. However, hope is insufficient and the change can only come with identification of the valid problems to be attacked and a conviction as to where the real design aims are coming from. Are they coming from the designer or are they a part of a set of myths? Based partially on reality but primarily having their roots in lack of understanding, lack of awareness of the user, and lack of that most essential of all ingredients, the willingness to look at a problem in a new way and understand that the world has changed, the real design aims must be examined carefully and before the system is unleashed on the innocent users.

Distributed intelligence for user-oriented computing

by TIEN CHI CHEN

IBM San Jose Research Laboratory
San Jose, California

INTRODUCTION

The primitives used by the computer designer have blossomed from the single logical connectives of two decades ago, into chips containing thousands of circuits and bits. Yet the quantitative aspect of the achievement, imposing as it is, signifies less than the qualitative injection of machine intelligence down to the chip level. With the consequent freedom to distribute computing power, machine design enters a new era.

We assert that very powerful extensible systems, based on the loose-coupling of nested autonomous modules, can harmonize with the hardware trends and be directed toward human-oriented, interpretive computing. The key to performance is self-optimization conducted throughout the polycentric system.

DISTRIBUTED INTELLIGENCE

Machine intelligence and self-optimization

An important aspect of intelligence is the ability to make, then follow, decisions based on available information. This ability is present in machines (in however small a measure) by the interplay among logic, memory, and the environment, especially through the use of stored programs.

Machine intelligence can be directed toward enhancing the processing throughput of the system itself; this self-optimization[1] can be practiced globally for the entire system, as well as locally within a unit of the system. The degree of self-optimization is largely dictated by cost and packaging considerations.

Local autonomy

In very large machines efficient processing cannot result from the complete prescription from a central mechanism. Local events with large performance implications may not be predictable, and it is too costly in time and bandwidth to submit these for central judgment in real time. It is clearly desirable to decentralize into intelligent autonomous units, each capable of local self-optimization.[1]

Until recently, local self-optimization has been difficult to achieve. Electronic circuitry was too expensive; inexpensive magnetic core memories are not only slow, but are package-incompatible. Designers had to evolve electronic devices with low circuit counts, in most cases barely achieving the minimum requirements for local intelligence.

A new era

The onrush of large-scale integration has now voided the technological distinction between logic and memory; their historical packaging incompatibility and speed mismatch have vanished. The cost of logic and memory have become very low, and is still dropping rapidly, and the size per circuit (or bit) also shrinks accordingly. A single chip (or a very small number of compatible chips) can now contain enough circuitry, non-volatile control memory, and read-write memory to work as a fast, low-cost unit with arbitrary degrees of stored-program character.[2]

With the revolutionary freedom to plant machine intelligence anywhere, machine design and machine usage should see a fundamental change. While the full implication of the new technology may not be fully known for some time, it is clear that the major stumbling block to effective local autonomy is no longer there.

The dawn of the new era brings new economies and new priorities. The production cost of hardware is measured mainly by the number of interconnections and the communication cost,[3] and the development cost is measured mainly by the number of chip types. Logical complexity has receded into secondary importance in the economic equation. Logic redundancy, no longer frowned upon, is becoming a practical necessity.

User accommodation

In these days of sharply lowering hardware costs and rising human values, no stone should be left unturned in accommodating the human user. The programming cost, now standing at roughly 40 percent of the total expenditure of average computer installations, stands in sharp contrast to the CPU cost, which is less than 10 percent of the total, and the overall hardware cost (30 percent).[4,5] It is now far more reasonable to bend the machine to suit the man than *vice versa*, and far easier.[6,7] Effective throughput analysis should take into account the entire problem-solving process.

Distributed intelligence

The new technology has already produced fast, small systems at attractive prices; it is relatively untried in the building of large supermachine systems.

While it is always possible to remap "proven" designs and mend the potential deficiencies in standard systems, the new capabilities, re-oriented priorities, and the increased appreciation of the users' role in global cost-performance strongly suggest that past designs may no longer be optimal.

We assert that very powerful, indefinitely extensible systems, harmonizing with new technology and new economic realities, can be constructed without the long development time which has been the bane of very large systems.

To transcend the basic switching time restrictions on performance in a supermachine, some form of multiprocessing is necessary. Our experience shows that intra-CPU communications tend to be vastly more frequent and more intricate than communications to the outside world (which could have other CPU's). It is therefore reasonable to base the communication-limited multisystem on the loose coupling among nearly self-sufficient, programmable modules. The structure then shows an extreme form of decentralization, not unlike a present-day computer network.[8] The system can be organized as a polycentric hierarchy, relying on labels and language for intercommunications.

Self-optimization will be instrumental to make such a decentralized system workable and efficient. An extension of the self-optimization scope can lead to optimal interpretation of higher-level languages, bringing genuine economy where it matters most, and reducing the barrier between man and machine.

Reappraisal

While reaching toward these goals a reassessment of the entire computing scene is necessary. Established techniques, conventions and viewpoints, their worth demonstrated in the past, may turn out to need revision.

Perhaps the greatest reward lies in merging concepts previously held to be distinct, irreconcilable opposites, but are more appropriately items for tradeoff. Examples are

logic-memory
hardware-software-firmware[9]
compilation-interpretation-execution
man-machine
control-cooperation

Actually, the walls keeping these concepts apart have been under erosion for some time; the new realities merely focus on their artificiality and hasten their demise. The barrier between logic and memory has already fallen; others will follow in due course.

THE NEED FOR LOOSE-COUPLING

Parallelism and pipelining

In discussing performance magnification, parallelism and pipelining invariably come to mind. Both are examples of tightly-coupled multiprocessing, demanding the intimate linking of a number of processors in a parallel or serial fashion, to give predictable behavior in detail. In a fully parallel system every functioning processing element moves in unison; in a fully pipelined system every microprocessor (pipeline segment) has a unit throughput, generating a result at the end of every cycle.

However, in general computing, these tightly-coupled designs cannot function efficiently without introducing loose-coupling and a measure of stored program character.

Diminishing returns

A job can be characterized by the parallelism ratio

$$\rho = \frac{\text{the amount of work which can be done in parallel}}{\text{the total amount of work}}$$

(a unit of work takes a single processor a quantum of time). With this definition it is easy to prove[10,7] that, regardless of the number M of parallel processors assigned for the job, the throughput, expressed as the effective number M_{eff} of fully utilized processors, cannot exceed $1/(1-\rho)$, which for most standard jobs ($\rho<0.9$) is less than 10 (see Figure 1). In fact

$M_{eff}=1$	for $M=1$	(efficiency 100%)
≤ 5	for $M=9$	(efficiency 55%)
≤ 9	for $M=81$	(efficiency 11%)
≤ 9.99	for $M=8991$	(efficiency 0.11%)

Pipeline systems are subject to the same law of diminishing returns (with ρ bounded by $(1-1/L)$, L being the number of similar jobs being sent through the pipeline). In realistic computations the efficiency loss is even more severe—conditional branches are notorious for serializing otherwise parallel processes, and for draining otherwise full pipelines.

Therefore, low though the cost for the added hardware may be, the unlimited use of parallelism and pipelining cannot be advised unless there is a scheme to fill the extra capacity. Otherwise the extra equipment would largely be wasted by disuse, except for special purpose computations.

Conversion to loose-coupling

Such an efficiency-inducing scheme is micro-multiprogramming, that is, multiprogramming adapted to the multiprocessing down to the smallest processing units.

Here the tight-coupling is relaxed sufficiently in the multiprocessors to accommodate unrelated pieces of work, so that idle processing power in executing a given task can gainfully be employed for some other tasks. The units in a parallel system, for example, will be called upon to do several unrelated tasks at the same time, and a pipelined system may be allowed extra entry-exit points in the interior. Then the input jobs are quantized for the available processing power, and the system must be preceded by a "throughput optimizer" consisting of a buffer and a selection mechanism to choose the appropriate tasks to maximize throughput, often deviating from the sequence of arrival of the task assignments.

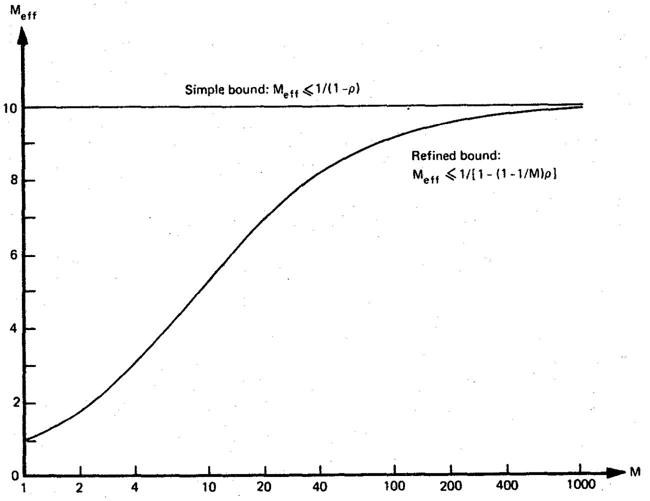


Figure 1—Bounds to tightly-coupled multiprocessor performance for $\rho \leq 0.9$

The tightly-coupled multiprocessor, thus converted for self-optimization, becomes loosely coupled to the environment, independent of the detailed timing and arrival sequence. It becomes a suitable module in a distributed intelligence system.

In such a system, the total workload may still be distributed by a central mechanism; but the local unit, no longer limited to passive processing, is now autonomous and will actively manipulate its own job queue for throughput. This way the administrative burden on the central mechanism will be lightened greatly—it is no longer necessary to know the detailed timing requirements of each unit before dispatching. Many more functional units can thus be brought under central "control".

The above analysis indicates that loose-coupling with self-optimization is a more viable form of multiprocessing than tight-coupling, which needs to be loosened to enhance throughput. While detailed predictability of sequencing is thereby lost, numerous techniques are available to guarantee the precedence of dependent events, hence the correctness of the final outcome. These interlocks again call for local intelligence to respond to messages accompanying the tasks.

POLYCENTRIC COMPUTING

Labels and routing

To reduce the inter-communication cost in a large system, processing power must be used to compensate for a limited connectivity and bandwidth, using a language-oriented identification scheme to resolve ambiguities and enhance processing.

An output from module A destined for module B cannot be specified by a physical path alone, with its multiple destinations; nor by time-slots because of the loose-coupling. The communication must rely on coded control, or label, using a common language of exchange. This coded control is needed not only to describe the information travel, but the environmental constraints accompanying the task, including the interlock information needed to preserve precedence as mentioned in the previous section.

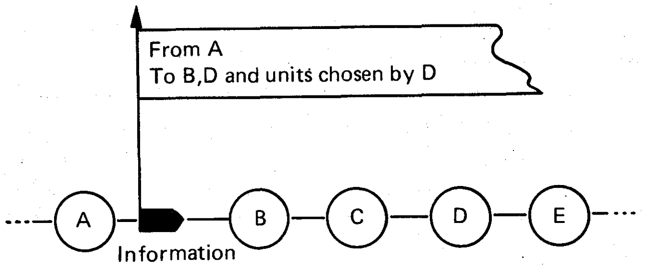


Figure 2—A routing slip

A job requiring processing by several modules in sequence can be given a "routing slip", specifying the major visitations, as shown in Figure 2.

We remark in passing that labels and routing slips already exist in many large machines (notably the S/360 Model 90 series[1]), computer networks, and the human society.

The routing need not always be encoded exactly, specifying all visitations in detail. The complete enumeration by a single agent may be expensive, or even impossible because of unforeseen or conditional events. It also tends to handicap the units downstream. Each unit should have limited freedom to determine both the subsequent routing and the ways to specify it; the full exercise of this freedom is an important aspect of self-optimization.

Typical modules

Most modules forming the nodes in the network can be classified conventionally into memory units, arithmetic units, instruction units, I/O channels, I/O units, and terminals.

Special modules may be necessary to direct the traffic, interpret the environment, and provide buffering in the event of congestion. These behave rather like intelligent I/O channels.

Loose-coupling demands the rather free use of computing power, especially non-numeric ability to interpret labels, and read-write associative memory to serve as buffers. For larger units it may be desirable to incorporate complete CPU-like stored program facilities for thorough self-optimization; the cost of redundancy in the new technology would be relatively low, and the possible gain in throughput and fail-safety may be considerable.

To provide the required power, a stored-program module should not be limited to hardware, distinguished by speed, bandwidth, memory and precision; it also may have software with its information complexity, high connectivity and flexibility, and firmware for efficient mapping into virtual machines. Distributed intelligence implies a complete dispersal of computing power throughout the system.

The modules can vary greatly in function, performance, and structural detail. Many may be identical; many may be complete minicomputers differing only in firmware and software. Some may be isolated units; the others will be constructed of smaller ones, like nested DO-loops in a Fortran program.

Polycentric nesting

The hierarchical nesting of modules form a polycentric system. At each level there is a collection of intelligent centers; each may have lower level intelligent centers as subsets, and so on down to an indivisible module. The entire polycentric system has the topology of a tree (see Figure 3).

Although there is a chain of command, the higher level commands to not exercise complete control over subordinates. While adhering to broad guidelines on responsibilities and system integrity, the low-level centers generally function as autonomous units to maximize throughput, and can process in a manner unexpected at the higher levels. Division of labor then can provide for the orderly flow of information, and provide a general framework for problem-solving. As stored program facilities are abundantly available, the labor-

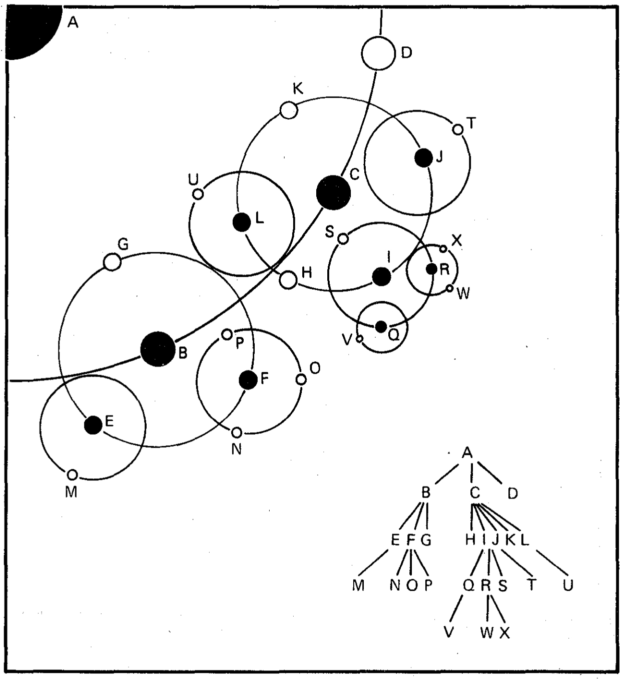


Figure 3—A polycentric system and its tree representation

division can be implemented within wide latitudes. The users' detailed needs are met by real-time assignments of subsets of the network.

The polycentric scheme offers many advantages. From the physical viewpoint, interconnections are simplified. Communication channels are partitioned into short, non-interfering pathways, each linking a small number of stations in good order (for a three level communication hierarchy see Pierce[11]). In an N-point system the path length between two arbitrary nodes of the tree is measured by log N, rather than N. The routing labelling is correspondingly systematized. Intracenter processing is naturally enhanced by the entailed short distances and easy access.

Further, entire centers can be units of replacement; addition, removal, or alterations can occur with least disturbance on the surroundings. The design and debug time of the large system is therefore expected to be much shorter than for more conventional systems.

The logical advantage to the designer lies in the clarity in delineating functional interdependence. There is no need to be bogged down at any stage by excessive detail. The structure offers a natural resilience; reprogramming (and remicroprogramming) can be practiced at all levels to compensate for design oversights, structural imbalances. Self-optimization, of course, is a type of reprogramming, dynamically rebalancing the system in real time.

Reductive mapping

A full logical specification of the polycentric system corresponds to a very large physical design. Feasible smaller physical designs can result from mapping onto fewer physical modules, trading performance for economy and compactness. Several logical modules within a center, sometimes even across center boundaries, may become "virtual machines" within the hardware, firmware, and software of one physical module, as shown in Figure 4. Modules may also undergo expansion to accommodate the increased workload.

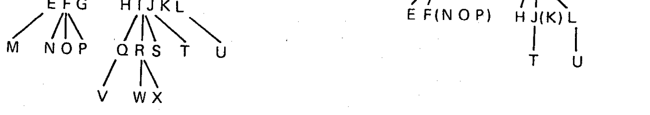


Figure 4—Reductive mapping

Examples of the reductive mapping may be:

1. The replacement of several stored program processors by one with a larger memory, and multiple sets of firmware;
2. Reduction of the degrees of parallelism, pipelining, and overlap;
3. Centralizing certain facilities, such as large memory and I/O channels, also sharing of one facility (say a fast multiplier) by several modules;
4. Rebalance the hardware-firmware-software admixture.[9]

When systematically conducted, the mapping should preserve the loose-coupling polycentric character.

Unified access

That the polycentric system may have more than one set of hardware, firmware, and software, and that entire modules may become "virtual", should not disturb the user; his problem is only the convenient access to computing power. The detailed means to provide power does not even concern the architect, whose job is to specify the logical mechanisms to provide service.

The system design is predicated upon a unified access to facilities, regardless of the implementation. For a given function, the allotment to particular unit(s) and the distribution among hardware, software, or firmware are all resource parameters subject to tradeoff by the designer of the particular systems. This tradeoff can even occur dynamically as an aspect of self-optimization and self-repair.

INTERPRETIVE PROCESSING

Nested programs and machine execution

It is now commonly agreed that a large program should be written as a nested collection of procedures with controlled interfaces. An important technique to generate nested procedures is top-down programming[12,13] based on the successive refinement of the original goal, which may be considered to be a zeroth level subgoal. A given subgoal may either be readily attainable, or, more commonly, may require the specification and attainment of next level subgoals. In programming, the successive goal refinement involves first the orderly creation of interfaces and environments for the lower level subprocedures, then the programming of the latter themselves in some order. The entire program has the topology of a tree.

Top-down programming maintains a clear framework at all times, sharply reducing the chances for misunderstanding and errors. The lower level programs are free to supply details once the interfaces with the higher levels are made explicit, and the programming can be assigned to relative strangers. The technique is a major ingredient in the "chief programmer team" approach to the management of software projects. The program produced this way is easy to read and maintain; modification also would occasion minimal disturbance to the rest of the program.[13]

The usefulness of nested programs would have been further enhanced, if their hierarchical structure is reflected during execution by a corresponding "top-down processing". In other words, the hierarchical program should ideally be handled by an "algorithm machine" which mirrors the program behavior in detail, so that the program never relinquishes control of the computation. The clarity in the top-down program will then be extended consistently into the execution phase. This feature will be particularly welcome in intimate man-machine interaction situations such as interactive computing and debugging, also in program modification during execution.

Very few experienced programmers today would believe that his own jobs are handled by a "Fortran machine". This is because most program executions are preceded by a compiling phase, when procedural code is replaced by machine code, concurrently "meaning" is replaced by representations. This way the original program syntax is beclouded; this is even more evident with object code optimization. There is a net information loss by compilation.

Compilation into machine code moreover tends to overspecify the processing, sharply limiting the possibilities of self-optimization. A case in point is the processing of a matrix innerloop by the IBM System/360 Model 90 series machines, which combine several machine instructions into a Fortran-like statement to permit efficient execution, thus undoing the compiling process.[1] The redundant use of intermediate registers in the compiled instructions turned out to be an overspecification and a major source of inefficiency there, and is avoided only by extra self-optimization effort.

Interpretative machines

This writer has advocated[10] constructing supermachines with some or all of the features below:

M: Multioperator statement execution, rather than instructions;

A: Array handling, rather than dealing with single numbers;

N: Name and descriptor handling, rather than addresses.

These features are mutually consistent, indeed mutually reinforcing. Together they form a framework for a truly interpretive machine, befitting the advance in large-scale integration.

Interpretive computing means top-down processing—the state of the interpretive machine can be arbitrarily close to the state of execution of the nested program, as is well-known to users of the (software) APL/360 interpretive system.[14] Recent experimental efforts to construct firmware and hardware interpretive machines[4,15–18] are encouraging responses to Gosden's challenge[19] that it is the engineers' role "to make processors and storage so fast, so large, and so cheap that interpreting becomes attractive."

We shall now briefly discuss each of these features in the light of the new technology and the polycentric system.

In advocating names and descriptors rather than addresses, let it be noted that the classical "address" is not really a tenable concept in the polycentric system. A piece of information may not have an address, if it is not yet available; it may have only a forwarding address if it has been moved; it may have several viable addresses if it has been copied, or several possible addresses if it is in transit. In many current large systems the address is already treated as a restricted label. As noted before, label processing is already needed with the linkage-constrained distributed intelligence system. The user names and descriptors may not exactly correspond with labels, but the technique and equipment for processing are essentially the same.

Array processing allows the proper scheduling of pipelines, parallel processors, and other tightly-coupled resources. The concise referral prevents piecemeal accesses. It frees the decoding mechanisms for decode-limited work elsewhere, and enhances high-bandwidth data flow. The ability to restructure arrays in real time avoids storage wastage and simplifies programming.

A statement is a concise quantum of procedure with a logical thread of causality, in sharp contrast to the fragmented steps in the compiled instructions. The direct use of statements simplifies routing assignments, protection, and interruption monitoring. The handling of intermediate results, no longer prescribed by instructions, is then subject to self-optimization.

Polycentric systems, being themselves hierarchical in nature, can further be designed to exploit the notion of successive refinement implied in the nested program. Clearly delineated subprocedures can be executed concurrently with little fear of conflicts. Each subprocedure can be allotted a "virtual machine" which in turn is

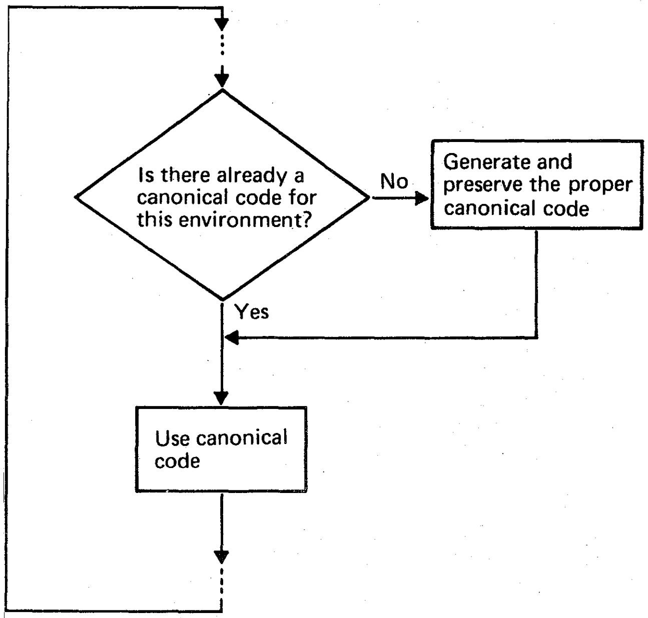

Is there already a canonical code for this environment?
No
Generate and preserve the proper canonical code
Yes
Use canonical code

Figure 5—Optimal interpretation

mapped into the polycentric system in real time. Such simultaneous assignments would be very difficult with conventional machine-language codes, because of the causality uncertainties.

Optimal interpretation

Interpretation, especially when performed exclusively by software, has a tendency to be slow, in contrast to the execution of a well-compiled Fortran object program. The performance gap may even widen, as compiling techniques to produce optimized machine code are constantly being improved.[20]

Optimal interpretation, though little known and seldom practiced, is just the real-time creation and selection of alternative processing techniques. It is a form of self-optimization, with stronger emphasis on linguistic manipulation and educated guesswork. There is also greater commitment of resources, particularly memory.

There are at least two different ways to specify the processing of a job, either in human-oriented procedural code, or in a more restrictive but machine-efficient "canonical" code. The latter can contribute significantly to efficiency for repetitious tasks. Therefore, the optimally interpretive machine can generate the canonical equivalent of the human-oriented code once, and use it at every known occurrence of the applicable environment. The original procedural code, being universal though slow, can be preserved for possible use whenever the available canonical code fails to apply. This way the top-down syntax is preserved until the system guarantees that equivalent results are obtained more efficiently.

A simple flowchart is given in Figure 5; the analogy to loop-traversal optimization having been pointed out in an earlier paper.[7]

For the APL language codes, Abrams[21] developed an optimal interpretation scheme to postpone array handling in anticipation of future simplifying actions, in order to minimize the time and memory cost of handling intermediate arrays. Here the choices are quite dissimilar: processing or procrastination, but the underlying principle is still the choice of alternatives. The Abrams scheme is also reminiscent of the self-optimizing mechanism to avoid redundant use of intermediate registers (as mentioned in an earlier section), but is far more elegant and powerful in the APL array processing context.

Self-optimization, conducted in real time, tends to favor tactics and ignore strategy, because of the limited scope and pressure of time. Nevertheless, strategic deployment of resources, heretofore the exclusive domain of human or compiler optimization, may be seen as a tactical issue at a high level for a well-nested hierarchical program, thus becoming exploitable by correspondingly hierarchical processing systems.

Potentially, optimized interpretation can reap most of the speed benefits of conventional compilation, without the syntax loss nor overspecifications of the latter. It combines advantages of incremental compilation, interpretation, and execution, and is in fact a merger of all three forms of processing.

CONCLUSION

In the polycentric system outlined, the distribution of machine intelligence becomes really practical only with the new technology.

In order to enhance system throughput, most of the processing resources are committed to work not explicitly specified by the user. Perhaps the greatest difference from previous decentralized systems will be in the pervasive use of self-optimization, that is, the exercise of machine initiative in real time. This occurs in queue selection to service tightly-coupled systems, choice of routing and label encoding in communications, dynamic trade-offs in function handling, and the creation and selection of equivalent problem-solving algorithms during optimal interpretation.

The price is a loss of detailed predictability—the user will not know how his job is handled. This is not a major issue; with a reliable computer system few users really want to know, and predictability already has been lost long ago with multiprogramming. Conceivably, the user might insist on doing things in the sequence he thinks best, and end up slowing down all processing, including his own.

Beyond choosing algorithms of increasing intricacy, a natural next step in the use of system initiative in interpretive computing would be to anticipate the user's needs, and to assist him in formulating his problem. This calls for very large systems with a sizable information library, and man-machine conversation near the natural language level. In the long run problem-solving and inquiry handling should become indistinguishable, and the supermachine need not be distinguished from a management information system.

ACKNOWLEDGMENT

Although the views expressed here in no way reflect those of IBM management, they result from the writer's exposure to various large IBM projects. The writer acknowledges stimulations received from his colleagues, especially Messrs. John C. McPherson, Carl J. Conti, William E. Worley, John R. Walters, Jr., Dines Bjørner, and Laszlo Belady. He is also indebted to Professors Harold Lawson, Wesley W. Chu and David J. Farber for helpful discussions.

REFERENCES

1 T C CHEN
The overlap design of the IBM System/360 Model 92 central processing unit
AFIPS Conf Proceedings FJCC 1964 Part II pp 73–83

2 H G RUDENBERG
Approaching the minicomputer on a silicon chip—Progress and expectations for LSI circuits
AFIPS Conf Proceedings SJCC 1972 pp 775–781

3 R RICE
Interaction between LSI and computer system architecture
Parallel Processor Systems, Technologies and Applications (L C Hobbs D J Theis J Trimble H Titus I Highberg editors) Spartan Books New York 1970 Chap 8 pp 165–190

4 R RICE W R SMITH
SYMBOL—A major departure from classic software-dominated von Neumann computing systems
AFIPS Conf Proceedings SJCC 1971 pp 575–587

5 S F DENNIS M G SMITH
LSI—Implications for future design and architecture
AFIPS Conf Proceedings SJCC 1972 pp 343–351

6 C McFARLAND
A language-oriented computer design
AFIPS Conf Proceedings FJCC 1970 pp 629–640

7 T C CHEN
Unconventional superspeed computer systems
AFIPS Conf Proceedings SJCC 1971 pp 365–371

8 D J FARBER
Networks: An introduction
Datamation April 1972 pp 36–39

9 H BARSAMIAN A DECEGAMA
Evaluation of hardware-firmware-software tradeoffs with mathematical modelling
AFIPS Conf Proceedings SJCC 1971 pp 151–161

10 T C CHEN
Parallelism, pipelining and computer efficiency
Computer Design Vol 10 pp 69–74 1971

11 J R PIERCE
How far can data loops go?
IEEE Trans Communications Vol COM-20 pp 527–530 1972

12 N WIRTH
Program development by stepwise refinement
Comm ACM Vol 14 pp 221–226 1871

13 F T BAKER
Chief programmer team management of production programming
IBM Systems J Vol 11 p 56–73 1972

14 L GILMAN A J ROSE
APL/360 an interactive approach
John Wiley and Sons New York 1970

15 A HASSITT
Microprogramming and high level languages
Proceedings 1971 IEEE Int Computer Soc Conf Sept 1971 pp 91–92

16 A HASSITT J W LAGESCHULTE L E LYON
Implementation of a high level language machine
4th Annual workshop on microprogramming Sept 1971 Preprints Volume To be published Comm ACM

17 R ZAKS D STEINGART J MOORE
A firmware APL time-sharing system
AFIPS Conf Proceedings SJCC 1971 pp 179–190

18 E W REIGEL U FABER D A FISHER
The interpreter—A microprogrammable building block system
AFIPS Conf Proceedings SJCC pp 705–723

19 J GOSDEN
Summary Session remarks (ACM Programming Languages and Pragmatics Conference Aug. 8–12 1965 at San Dimas, Calif)
Comm ACM Vol 9 p 220 1966

20 F E ALLEN
Program optimization
Annual Review in Automatic Programming Vol 5 1969

21 P ABRAMS
An APL machine
PhD thesis Stanford University 1970

Design of a fault-tolerant, modular computer with dynamic redundancy*

by RALPH B. CONN and NIKITAS A. ALEXANDRIDIS

Ultrasystems, Incorporated
Newport Beach, California

and

ALGIRDAS AVIŽIENIS

University of California, Computer Science Dept.
Los Angeles, California

INTRODUCTION

This paper presents the results of a design study for a Modular Spacecraft Computer (MSC). The MSC is a fault-tolerant computer that is designed to preserve the continuity and correctness of its programs in the presence of both transient malfunctions and permanent failures in its hardware. The MSC is designed to detect and to implement recovery from faults without external assistance. The fault-tolerance goals are to provide a reliability of 0.95 for a five-year period of unattended operation and to provide recovery from a specified class of repetitive transient malfunctions, bounded by maximum repetition rate and maximum duration specifications, during that time period.

Dynamic redundancy (self-repair)[1] is used in most parts of the MSC to achieve fault-tolerance. The reliability prediction is based on a specific choice of logic and memory technology. The MSC is required to provide a constant computational capability over a five-year period, therefore partial fault-tolerance ("graceful degradation") is not applicable in this case. The required reliability is attained by the use of spare modules and continuous ("concurrent") fault-detection procedures that verify the execution of every instruction before the next instruction is executed. Errors that are due to transient faults are eliminated by "rollback", i.e., repetition of a specified sequence of the current program. Replacement of failed modules is employed in the case of permanent faults, then a "rollback" is used to restart regular operation.

The foundations for this study have been supplied by recent research on fault-tolerant computer design, mainly the JPL STAR computer,[2] the IBM MARCS design,[3] and studies at the MIT C S Draper Laboratory.[4] The goal was to define a fault-tolerant architecture that would meet a set of performance and reliability specifications, and physical constraints (especially weight and power), using projected logic and memory technologies. The detailed design led to the identification and resolution of several fault-tolerance problems. One major result is the demonstration that the level of modularization depends very strongly on the hardware technology, and that the critical factors at the present time are the reliability and physical characteristics of non-volatile random-access memory modules. Other major results are: a detailed functional design of a Configuration Control Processor (CCP), methods to implement the replacement of Power Converter and Timing (clock) Modules, development of a class of fault-tolerance aiding instructions, a systematic CCP-replacement procedure, use of spare control bus wires, and error-coded floating-point arithmetic.

The system organization and the reliability modeling of the MSC are presented in the following sections. The design approach chosen is that of functional modularization[2,3] with extensive use of error-detecting codes,[5] rather than the partitioning into "subcom-

* This work was supported by the Space and Missile Systems Organization of the U.S. Air Force under Contract F04701-71-C-0361.

puters" operating in a multiprocessing mode with complete duplexing.[4] Reliability modeling is illustrated by considering a number of competitive MSC configurations in order to arrive at a most cost-effective choice. The results show that the implementation of complete fault-tolerance and self-repair in a spacecraft computer is feasible within the constraints of projected logic and memory technologies.

MSC SYSTEM ORGANIZATION

Modularization

The baseline design presented is that of a fully-parallel, fault-tolerant, modular spacecraft computer. The recommended design evolved from consideration of four different possible MSC architectures. The initial architecture, which is called the "functionally modular MSC" (see Figure 1) used the most detailed level of modularization into functional modules. The architecture of the functionally modular MSC was outlined in detail and estimates made of the required equipment. The detailed architecture was formulated for a fully-parallel (36-bit words) MSC and two versions using byte-serial information transfer of 4 and 12 bits respectively. Finally, an alternative fully-parallel design called the "combined processor MSC" was evolved. The following discussion provides the background for the comparison and choice that was made.

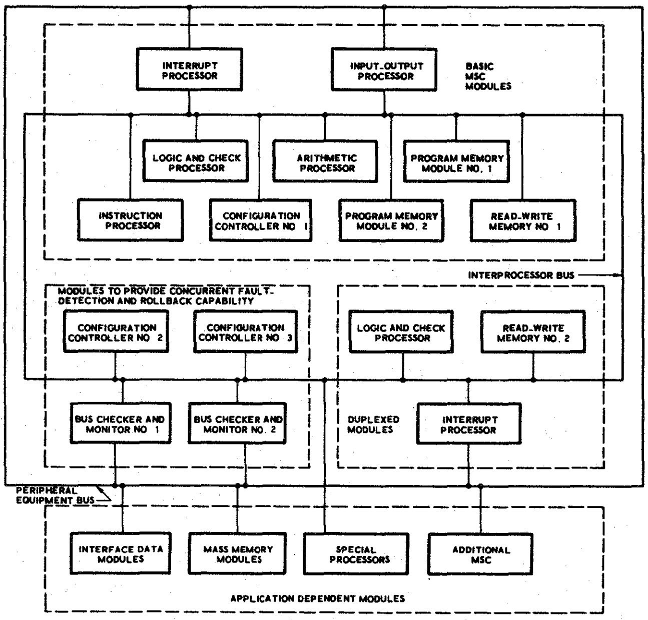

INTERRUPT PROCESSOR
INPUT_OUTPUT PROCESSOR
BASIC MSC MODULES
LOGIC AND CHECK PROCESSOR
ARITHMETIC PROCESSOR
PROGRAM MEMORY MODULE NO. 1
INSTRUCTION PROCESSOR
CONFIGURATION CONTROLLER NO 1
PROGRAM MEMORY MODULE NO. 2
READ_WRITE MEMORY NO 1
INTERPROCESSOR BUS
MODULES TO PROVIDE CONCURRENT FAULT-DETECTION AND ROLLBACK CAPABILITY
CONFIGURATION CONTROLLER NO 2
CONFIGURATION CONTROLLER NO 3
LOGIC AND CHECK PROCESSOR
READ_WRITE MEMORY NO. 2
BUS CHECKER AND MONITOR NO 1
BUS CHECKER AND MONITOR NO. 2
DUPLEXED MODULES
INTERRUPT PROCESSOR
PERIPHERAL EQUIPMENT BUS
INTERFACE DATA MODULES
MASS MEMORY MODULES
SPECIAL PROCESSORS
ADDITIONAL MSC
APPLICATION DEPENDENT MODULES

Figure 1—MSC block diagram—Functionally modular approach

Functional modularity of the initial MSC design was achieved through use of the following techniques. First, the processors are divided functionally. Arithmetic is performed in the arithmetic processor, logic operations in the logic processor, etc. Much of this modularity was retained in the recommended baseline design when, as a result of reliability and physical characteristic analyses, some functional processors were combined. Similarly, the memories are divided functionally into a read-write or scratch-pad memory and the program memory. The memories are further divided into modules. Modularity of operations rate is achieved through variation of bus width and/or clock rate. Adaptability to various peripheral devices is achieved through use of input-output processors of various speeds, and through use of a peripheral-device connection bus.

Figure 1 has been arranged so as to illustrate the basic configuration of a non-redundant ("perfect") modular computer, and then to show the additional equipment required to make the machine fault-tolerant, and also to identify the application-dependent modules. Spares and power, timing, and control buses are not explicitly shown. The upper nine rectangles enclosed by the dashed line are the modules used in the functionally modular MSC design. In addition (not shown) there are timing and power converter modules. The instruction processor (ISP) performs the instruction counter manipulation both for normal sequencing and in connection with transfer operations. The index registers are part of the ISP and the operations associated with these are executed by the ISP. The logic and check processor (LCP) performs all logic instructions and also checks all bus transmissions for correct coding. The arithmetic processor (ARP) executes the normal arithmetic instructions. The input-output processor (IOP) provides data buffering and control signals to the peripheral units. The interrupt processor (IRP) provides storage for received interrupt signals and executes the instructions concerned with interrupt handling. The read-write memory (RWM) holds the input data, the temporary results of computations, and the results to be output. The program memory (PRM), operated in a read-only manner, holds the instructions of the program. Two modules of PRM are needed to contain all programs. All of the above units are connected to the interprocessor bus, as is the configuration control processor (CCP) which performs a function similar to that of the instruction timing unit in an ordinary computer.

However, the MSC is not an ordinary computer, but rather a fault-tolerant computer employing concurrent fault detection. The principal method of fault detection is through the use of error detecting codes applied to each word. The error code is preserved through arithmetic operations. Non-numerical operations, e.g., logic operations, do not preserve these check codes and thus the LCP and IRP modules are duplexed. The read-write memory module is duplexed in order to retain a correct copy of information in case of memory failure. Error detection for duplexed modules is accomplished by checking for disagreement between the outputs of two corresponding modules. When a disagreement is detected, the error detecting code is used to quickly isolate the faulty module. The duplexed modules are shown on the right side in Figure 1.

Fault detection is aided and rollback capability made possible through the use of the configuration control processor (CCP). Because of the critically important functions of this processor, it is operated in triplicate with voting at the outputs. The two additional CCP modules necessary for voting are shown here, as is a duplexed bus checker and monitor (BCM) module, which performs the checking and monitoring function on the peripheral equipment data bus. It is connected to the interprocessor bus for communication with the input-output processor and the CCP. Finally, the units at the left of Figure 1 are the application-dependent modules. The mission data modules, the mass memory modules, and the additional MSC (if required) are all connected to the peripheral equipment bus. The special processors, which may be added to compute special functions, from generation of a square root to a fast-Fourier transform are connected to the interprocessor bus.

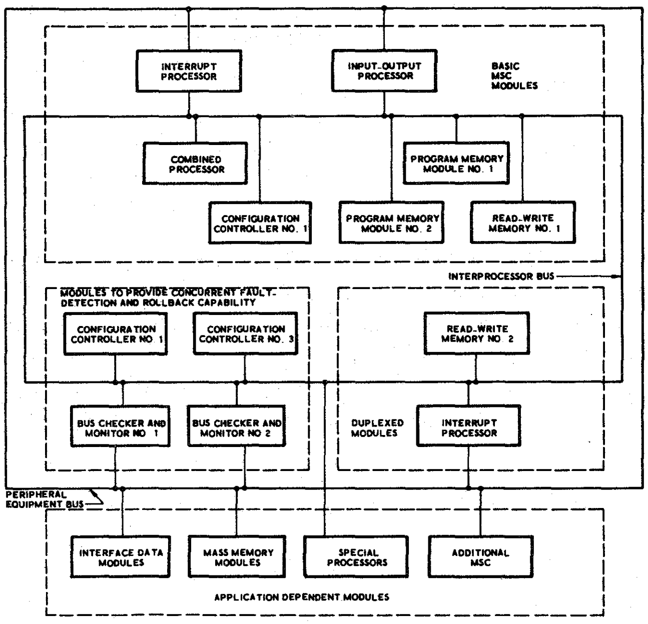

INTERRUPT PROCESSOR
INPUT-OUTPUT PROCESSOR
BASIC MSC MODULES
COMBINED PROCESSOR
PROGRAM MEMORY MODULE NO. 1
CONFIGURATION CONTROLLER NO. 1
PROGRAM MEMORY MODULE NO. 2
READ-WRITE MEMORY NO. 1
INTERPROCESSOR BUS
MODULES TO PROVIDE CONCURRENT FAULT-DETECTION AND ROLLBACK CAPABILITY
CONFIGURATION CONTROLLER NO 1
CONFIGURATION CONTROLLER NO. 3
READ-WRITE MEMORY NO 2
BUS CHECKER AND MONITOR NO 1
BUS CHECKER AND MONITOR NO 2
DUPLEXED MODULES
INTERRUPT PROCESSOR
PERIPHERAL EQUIPMENT BUS
INTERFACE DATA MODULES
MASS MEMORY MODULES
SPECIAL PROCESSORS
ADDITIONAL MSC
APPLICATION DEPENDENT MODULES

Figure 2—MSC block diagram—Combined processor approach

TABLE I—Baseline Configuration Characteristics

	Basic Quantity	Spares
Power Converter Module	1	2
Timing Module	1	2
Input-Output Processor	1	3
Interrupt Processor	2*	3
Program Memory	2	4
Read-Write Memory	2*	5
Combined Processor	1	3
Configuration Control Processor	3**	1
Five-Year Reliability	0.953	
Power	21 watts	
Volume	582 cubic inches	
Weight	33 pounds	

* Operated in duplex
** Operated in triplex with voting

With this detailed architecture available it was then possible to make estimates of processor reliabilities and physical characteristics. An analysis of these results illustrated that the modularization overhead, for the projected technologies, was high enough to question the idealized theoretical conclusion that increased partitioning leads to greater reliability.[6] Thus a new MSC design was formulated, which employed parallel bus transfer and combined the functions of three of the processors (ISP, LCP, ARP) of the functionally modular approach into a single combined processor CP. The block diagram of the Combined Processor MSC is shown in Figure 2 and it is the recommended baseline design. Because of the detailed architecture that had been formulated for the functionally modular approach, it was possible to make this combination quickly and accurately without repeating the design work associated with each processor. It must be noted that the "functionally modular" approach of Figure 1 also provides the required reliability and that it may become superior to the "combined processor" approach if the evaluation of component technologies departs from the current projections.

Table I summarizes the properties of the baseline MSC configuration, including the number of spares needed to attain the required five-year reliability. The projected physical characteristics are also presented.

Functional characteristics

A summary of the principal MSC characteristics (excluding the fault-tolerance features) is presented in Table II. The most unusual feature in this table is the use of check bits in both the data words and instructions of the MSC. The formats of MSC words are shown in Figure 3. The error-detecting code used in the MSC is the inverse residue code, using the check modulus 15. The four-bit "check byte" C(X) is attached to a word which represents the integer value X. It has the value:

$$X(X) = 15 - 15|X$$

where 15|X designates "the modulo 15 residue of X." The floating-point word has separate check bytes for its two parts. The instruction word uses a 16-bit address part that is divided into a 4-bit "module name" part and a 12-bit "internal address" part for addressing within the module. Separate check bytes are provided for both parts of the address, as well as for the 8-bit operation-code part.

It is very important to note that the "module name" in the address part refers to the "soft" name of a memory module. Every memory module (including all spares) has its unique "hard" name. When power is first turned on in a memory module, a special instruction "Set Soft Name" is employed to assign a four-bit "soft" name to the module. All other instructions refer to this "soft" name only. Duplex operation is attained by assigning the same "soft" name to two memory modules. A failed module is replaced by assigning its "soft" name to a newly powered spare; thus continuity of addressing is preserved although replacement has taken place.

The instruction set of 58 basic operation codes has a complete repertoire of control, indexing, logic, arithmetic (fixed and floating point), interrupt-handling and I/O operations. In addition, there are five special "fault-tolerance class" operation codes which include "Set Soft Name" discussed above and four other operation codes (to be discussed in the following section).

TABLE II—MSC Functional Characteristics

Number System	Binary, fixed-point (floating point option provided)
Data Representation	Fractional two's complement for fixed point Fractional two's complement coefficient } For floating point Integer exponent (excess—32) } For floating point System radix—16 } For floating point
Information Transmission	Parallel
Data Word Length	Fixed point—32 data bits plus 4 check bits Floating point { 22 coefficient bits plus 4 check bits 6 exponent bits plus 4 check bits
Instruction Word Length	36 bits including 12 check bits
Instruction List	90 single-address instructions using 58 basic operation codes which include special fault-tolerance operation codes
Indexing Capability	3 20-bit, plus 4 check bits, index registers
Input/Output Control	Simultaneous single input or output with compute
Interrupt Capability	Accepts up to 32 external interrupts, each maskable under program control.
Memory Modularization	Program memory is modular in 4K, 36-bit word modules. Read-write memory is modular in 2K, 36-bit word modules.
Memory Capacity	Maximum directly addressable memory (program plus read-write) is 63,488 words, or 16 modules. The allowable number of spare memory modules is not restricted. Program memory modules can be operated in duplex. Baseline capacity is 8K words of program memory and 2K words of duplexed read-write memory.

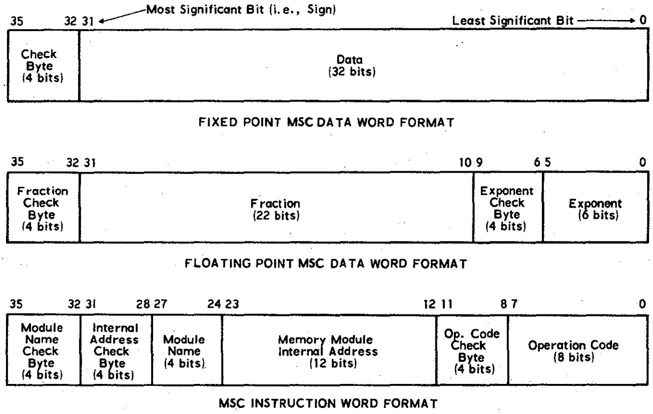

Most Significant Bit (i.e., Sign)
35
32 31
Least Significant Bit
0
Check Byte (4 bits)
Data (32 bits)
FIXED POINT MSC DATA WORD FORMAT
35
32 31
10 9
6 5
0
Fraction Check Byte (4 bits)
Fraction (22 bits)
Exponent Check Byte (4 bits)
Exponent (6 bits)
FLOATING POINT MSC DATA WORD FORMAT
35
32 31
28 27
24 23
12 11
8 7
0
Module Name Check Byte (4 bits)
Internal Address Check Byte (4 bits)
Module Name (4 bits)
Memory Module Internal Address (12 bits)
Op. Code Check Byte (4 bits)
Operation Code (8 bits)
MSC INSTRUCTION WORD FORMAT

Figure 3—MSC word formats

FAULT-TOLERANCE FEATURES OF THE MSC

Approach

Dynamic redundancy (self-repair) was chosen as the main fault-tolerance method in the MSC. The

TABLE III—MSC Fault-Tolerance Features

Modularity	The computer is composed of automatically replaceable processor, memory, timing, and power supply modules.
Concurrent Diagnosis	Fault-detection is performed concurrently with instruction execution through use of inverse, modulo-15, residue code and module status messages.
Duplex Operation	An option is provided in which every type of processor or memory module may be operated in duplex mode as long as spares exist.
Data Redundancy	Redundant data storage is provided in duplexed read-write memory modules.
Self-Repair	Permanently failed processor, memory, timing, and power supply modules are automatically replaced by spares. The allowable number of spares is not restricted by the design and may be adjusted to mission reliability requirements.
Program Restart	Automatic program restart, on fault detection or after module replacement, to programmer-specified restart point.
System Restart	Automatic system restart following occurrence of catastrophic faults.
Power Switching	A module is removed by turning its power off, and connected into the system by turning power on. Unpowered modules are connected to the bus, but produce only logic "zero" outputs.
Centralized Monitoring	The Configuration Control Processor monitors computer operations by observing status messages and by checking the code validity of all words being sent by the bus. Hybrid redundancy (triplication with voting and with spares) is employed to provide CCP fault-tolerance. The CCP executes all restart and replacement operations.

principal features of the MSC fault-tolerance approach are summarized in Table III. The design utilizes a number of features which were proven in the experimental STAR computer,[2] as well as the self-checking logic circuits of the MARCS study[3] to protect the critical fault-detecting logic within the MSC modules.

Four fault-tolerance instructions that serve to augment the hardware monitoring are: Test Fault Signal (TFS), Power Control (PWC), Read Status Word (RSW), and Update Restart Register (URR). TFS is employed to verify that fault signals in a designated module are still operational. PWC designates a module in which power is to be turned off or on. This allows program-controlled checkout of spares, change to duplexed operation, and increase or decrease of available memory. RSW reads out a status word that is internally stored in every module, and indicates the cause of a fault message to the CCP. URR causes its address part to be stored within the CCP in the register that contains the address at which the program is to be resumed if a fault is detected.

Fault detection

This section presents the methods used to check data and instruction words and to identify faults in the different parts of the MSC.

Interprocessor bus (IPB)

All information transmitted on the bus is checked for code validity by the "Bus Checker" in the CCP. Since the CCP also accepts and decodes instructions and issues the synchronization signals (T1, T2, . . . T6) to the various MSC modules it knows at each instant what type of information (data vs. instructions) is placed on the bus. For data words, it performs a modulo-15 addition of the nine 4-bit bytes. For instruction words, it performs modulo-15 additions of the various fields that contain their own check-bytes. (See Figure 3)

Combined processor (CP)

There are four facets of operation code validation in the CP.

a. The 6-bit operation code, the 4-bit check-byte (both in the operation code register), and the two index bits (held internally in the referenced index register) are all added in a modulo-15 adder.
b. Each microinstruction is parity checked.
c. Microinstruction bits are compared with "Reply Back" bits from activated gates.
d. The 6-bit operation code is compared with the 6-bit operation code stored in the last microinstruction of the corresponding microprogram.

A modulo-15 adder is used to: (a) check the effective address calculation (i.e., indexing and instruction-counter incrementing); (b) check the interim results of arithmetic operations; (c) generate the checkbyte for a 32-bit logic results; (d) monitor bus coupler failures by checking the operand entering the CP.

The logic operations hardware (e.g., Logic Operations Net, Shift Net, etc.) is duplicated and the independently obtained results are compared for equality. Any mismatch is reported to the CCP. Two logic accumulators and two operand registers are provided to identify possible faults at the inputs of these duplexed logic nets.

The CP internally-held word is compared with the word placed on the bus by the CP to identify failures of the CP output circuits.

Input-output processor (IOP)

Data words input from the IOP are placed on the interprocessor bus which is monitored by the CCP while IOP output is placed on the peripheral equipment bus which is monitored by the Bus Monitor and Checker (BMC). Program and scratchpad memory addresses are checked by the CCP and also inside the memory modules (for Read-Write). The IOP's "Data Block" counter is monitored by the CCP. Operation code validation is accomplished as in the CP.

Interrupt processor (IRP)

Two IRP's are operated in duplex. Interrupt signals are transmitted on the interrupt bus encoded in "one-out-of-two" code and are internally translated from double-rail[3] to single-rail signals prior to their being sent to the Interrupt Register. A modulo-15 generates the 4-bit checkbyte for the 32-bit Interrupt Register. Results of IRP operations are placed on the interprocessor bus. These are checked by the "Bus Checker" in CCP. Proper functioning of an IRP module is checked by comparing the internally held result with the word appearing on the bus. Operation code validation is accomplished as in the CP.

Read-write memory modules (RWM)

RWM modules are always operated in duplex and their internally held words are compared with the word placed on the IPB. Each memory module in a functioning RWM duplexed pair has a unique 4-bit Hard Name, but both have the same assigned 4-bit Soft Name. The CCP determines whether the module with the correct "Hard Name" responds. In addressing a module, the module name (MN, MN′) of the received address is compared with the addressed memory module's assigned soft name (SN, SN′). If they match, then the Internal Address (IA) decoder is enabled. The received internal address (IA, IA′) is used for addressing within the module. Each word in a memory module has four additional read-only bits appended to it. These bits are the inverse, modulo-15 residue code of the word address (WA). During "Read," IA′ (the 4-bit checkbyte for the internal address field IA) is compared with WA′ to verify that the proper location has been accessed. "Write" is preceded by the comparison of IA′ and WA′ to avoid overwriting and destroying a word in the wrong location. To assure that a "Write" took place in the correct modules, an EVENT status message is sent to the CCP identifying the responding modules' Hard Names after the "Write" has taken place.

Program memory modules (PRM)

PRM modules operate in simplex, Read-Only mode. "Write" circuitry is disabled for ordinary operation (it is enabled only during loading from mass memory). Monitoring and checking is accomplished às in the RWM.

Timing module (TM)

To provide detection of wave shape deterioration, internal duplexing and comparison of oscillator outputs was chosen. The TM is replaced when internal discrepancy is detected and reported to the CCP. The CCP's contain independent internal timing (provided by a delay line) which serves to bridge the TM switchover time.

Power converter module (PCM)

The CCP contains a Power Monitor circuit which issues a Power Alarm signal to CCP when a PCM voltage strays out of tolerance (too high or too low). Replacement of a faulty PCM is accomplished under CCP control.

Configuration control processor (CCP)

Triple redundancy with voting is used for the CCP outputs. When one CCP disagrees, a transient error is assumed, and a "circulate CCP registers" is performed serially through a voter. If disagreement reappears, the faulty CCP is replaced by power switching. Every CCP compares the voted output with its own input to the voter. In case of a disagreement, the CCP issues a "distress" message to the other CCP modules, which initiate a CCP register circulation sequence. If the "distress" message reappears, the two good CCP modules switch off power to the bad CCP, switch in a spare, and once again go through the CCP register circulation sequence in order to load the new "blank" CCP with all necessary information.

INSTRUCTION EXECUTION

This section introduces the concurrent instruction sequences in the MSC processors. An arithmetic-type instruction with memory operand access has been selected as an example and its sequence is shown in Table IV. Letter-Gothic type has been used for the ordinary computer actions and italic type for the fault-tolerance (error detection) actions. Fetch-execute overlap is now shown in this table and the IOP and IRP are omitted.

The events occurring during each time interval are described below.

Ta: The CP issues the instruction counter (IC) which is accepted by the PRM. The Module Name field of the IC is also accepted by the CCP.

Tb: CP increments IC and checks this operation and the PRM accesses the next instruction. CCP decodes the module name field of the IC and uses a hardware table to identify memory modules whose hard name has been assigned a soft name equal to the IC module-name field.

Tc: PRM issues next instruction which is accepted by all MSC modules. The Hard Name of the responding PRM module is sent to the CCP which in turn identifies whether the correct memory module responded.

Td: Indexing (if necessary) is performed in the CP, while CCP checks the accepted next instruction for validity.

Te: CP issues the operand address (on the bus) which is accepted by CCP and RWM.

Tf: This address is checked for validity in the CCP and is used by the RWM to access the operand.

Tg: RWM issues the operand (on the bus) which is accepted by the CP and CCP.

Th: The operand is used in the CP to compute the arithmetic result and is at the same time checked for validity in the CCP.

Ti: The arithmetic result (including computed check bits) is placed on the bus and accepted by the CCP.

Tj: The CCP checks the result check byte to detect failures in the CP.

Overlap of present-instruction execution and next-instruction fetch was implemented in the MSC baseline as shown in Table V to increase the computational rate of the system. The SYNC signals (T1, T2, . . . T6) are operation code dependent and are issued by the CCP to all units so that they can advance in synchronism to the next sequence step. Each instruction type is identified by CCP to determine the time duration of each SYNC signal (e.g., for a DIVIDE instruc-

TABLE IV—Arithmetic Instruction (With Operand) Sequence

Sync Signals	Combined Processor (CP)	Configuration Control Processor (CCP)	Program Memory (PRM)	Read-Write Memory (RWM)
Ta	Issue IC	*Accept Module Name*	Accept IC	
Tb	Increment IC *(check IC incrementing)*	*Decode Module Name to Identify Hard Name*	Access Next Instruction	
Tc	Accept Next Instruction	Accept Next Instruction *Correct Hard Name Responded?*	Issue Next Instruction *Send Hard Name to CCP* Accept the issued instruction	Accept Next Instruction
	Process Instr. Address Part (i.e. indexing, if required)	*Check Next Instruction*		
Te	Issue Indexed Address	*Accept Indexed Address*		Accept Indexed Address
Tf		*Check Address*		Access Operand
Tg	Accept Operand	*Accept Operand*		Issue Operand
Th	Compute Arithmetic Result	*Check Operand*		
Ti	*Issue Arithmetic Result*	*Accept Arithmetic Result*		
Tj		*Check Arithmetic Result*		

TABLE V—Arithmetic Instruction Sequence With Overlap

Sync Signals	Combined Processor (CP)	Configuration Control Processor (CCP)	Program Memory (PRM)	Read-Write Memory (RWM)
T5	Process Instr.(n) Ad. Part (indexing, if required)	*Check Instr. (n)*		
T6	Issue Indexed Ad. (n)	*Accept Indexed Ad. (n)*		Accept Indexed Ad. (n)
T1	Issue IC (n+1)	*Accept IC (n+1)* *Check Indexed Ad. (n)*	Accept IC (n+1)	Access Operand (n)
T2	Increment IC (to n+2) (*check incrementing*) Accept Operand (n)	*Accept Operand (n)* *Decode Mod. Name of IC (n+1) to Identify Hard Name* *Check IC (n+1)*	Access Next Instr. (n+1)	Issue Operand (n)
T3	Compute Arith. Result (n)	*Check Operand (n)*	[continue access]	
T4	Accept Next Instr. (n+1)	Accept Next Instr. (n+1)	Issue Next Instr. (n+1) Accept Issued Instr. (n+1)	Accept Next Instr. (n+1)
T5	Process Instr. (n+1) Ad. Part (indexing, if required) *Issue Arith. Result (n)*	*Check Next Instr. (n+1)* *Accept Arith. Result (n)*		
T6	Issue Indexed Ad. (n+1)	*Accept Indexed Ad. (n+1)* *Check Arithm. Result (n)*		Accept Indexed Ad. (n+1)

tion, CCP would issue T4 much later than it would for an ADD).

RELIABILITY MODELING

The objective of the MSC reliability analysis and calculations was, in conjunction with appropriate physical requirements determinations, to answer the questions, "What should be the MSC baseline configuration?" and "How many spares are required for the desired reliability and mission time objectives to be met?" A related consideration is the sensitivity of the results to the assumptions used. The approaches taken to obtaining these answers, the computational results, and their discussion are presented in the following three subsections.

Approach

The basic approach to the computation of the MSC reliability was to determine the reliability of the MSC modules and then, for the assumed number of spares, the power-on/power-off failure rate ratio, and coverage, to calculate the MSC reliability. The reliability models for the two contending MSC configurations are implied in the information presented in Table VI.

The CCP units are operated in a hybrid-redundant fashion. That is, the three units with power-on are operated in a TMR configuration, but with standby spares for replacement when one of the active units fails. The RWM, IRP, and LCP units are operated in duplex for checking purposes. It is assumed that when one of the last two units fails, that the module group has failed. This is the most conservative approach. The remaining units are all operated in simplex standby redundancy.

Since every group of modules must survive the mission, it is evident that the MSC reliability is the product of the module group reliabilities. These module group reliabilities are a function of the module failure rates, the power-on/power-off failure rate, the coverage, and the number of spares utilized.

TABLE VI—Operating Modes For MSC Units

	Operating Mode	
Unit	Functionally Modular Design	Combined Processor Design
CCP	TMR	TMR
RWM	Duplex	Duplex
IRP	Duplex	Duplex
LCP	Duplex	Not Used
PRM	Simplex	Simplex
IOP	Simplex	Simplex
TM	Simplex	Simplex
PCM	Simplex	Simplex
ISP	Simplex	Not Used
ARP	Simplex	Not Used
CP	Not Used	Simplex

Failure rate determinations

Failure rates for LSI arrays are not available in the published component reliability tabulations and therefore must be estimated. The approach taken[7,8] involves establishing the failure rate for an integrated circuit of quality and complexity comparable to the basic cell of the large scale array. This failure rate is then apportioned according to the failure modes contributing to the failures. The failure mechanisms which cause each mode of failure are then grouped according to the failure modes contributing to the failures. The failure mechanisms which cause each mode of failure are then grouped according to the point in the processing where they are introduced and weighted to reflect the frequency of occurrence. The fractional failure rates are then weighted to reflect the differences between integrated circuit and LSI technologies. The summation of these weighted fractional failure rates is then the LSI failure rate. This base failure rate is then modified by complexity, package type, environmental, and quality factors. Assuming 300 gates and 40 connections per chip, a chip failure rate of 0.0985 failures/10^6 hours was obtained. This figure agrees quite well with those implied in Reference 9. Failure rates for other components were estimated using data from RADC-TR-67-108.[10] The resulting module failure rates are shown in Table VII.

Reliability computations

The reliability computations reported below were performed using the Computer-Aided Reliability Estimation (CARE) program developed at the Jet Propulsion Laboratory.[11] The program was modified to delete the extensive plotting capabilities and to improve the printout format.

TABLE VII—MSC Module Failure Rates

Module	Module Failure Rate (λ) (Failures per million hours)
Configuration Control Processor	2.1
Read-Write Memory	7.8
Logic and Check Processor	2.5
Interrupt Processor	2.4
Program Memory (per 4K module)	10.2
Instruction Processor	2.9
Arithmetic Processor	3.4
Combined Processor	5.6
Input-Output Processor	5.1
Timing Module	0.5
Power Supply	1.8

There are two types of reliability configurations used in the MSC design. These are the hybrid-redundant configuration used for the CCP, and the standby-replacement configurations used for all other modules. The applicable equations are given in Reference 11.

CARE produces, as a function of time, the reliability of each module, the reliability of the module group for the assumed number of spares, and the reliability of the entire system, i.e., the MSC.

The reliability computations were performed for four MSC configurations. These configurations are the: Functionally modular, fully parallel; combined processor, fully parallel; functionally modular, 12-bit byte-serial; and, functionally modular, 4-bit byte-serial.

Reliability tradeoffs

The data obtained showed that for all cases the five-year reliability of the MSC(CP) is higher than that of the MSC(FM). However, in none of the cases, is the advantage for the MSC(CP) so overwhelming as to make it the obvious choice. Perhaps the most significant conclusion that can be drawn from the data is that the modularization overhead is high enough to keep from disqualifying the MSC(CP) altogether. This in itself is a significant conclusion since past studies which ignored the modularization overhead concluded that the greater the modularization, the more dramatic the life improvement. A second significant conclusion is that there are configurations for coverage values of 0.99 that exceed the desired 0.95 five-year success probability. The importance of knowing what coverage is achieved by a fault-tolerant computer design, and achieving high coverage, was demonstrated by assuming a coverage of 0.95. Even for high sparing and the power-on/power-off failure rate ratio (K) of 2, the five-year reliability is below 0.88.

Before attempting to resolve MSC(FM)/MSC(CP) question it is necessary to examine systems that are designed to meet the 95 percent mission success probability criterion.

The five configurations shown in Figure 4 use K equal to one for the PS, TM, PRM, and RWM, and K equal to two for the remaining modules. While there are undoubtedly those who might wish to relax K even further, this combination is thought to be a reasonable compromise. The initial two configurations of Figure 4 assume coverage equal to $0.\overline{9}$, while the remainder use the individual "best estimate" values of

```
UNIT         PS         TM         IOP        IRP        PRM        RWM        ARP        ISP        LCP        CCP        CP
MODE      SIMPLEX    SIMPLEX    SIMPLEX    DUPLEX     SIMPLEX    DUPLEX     SIMPLEX    SIMPLEX    DUPLEX     H(3.S)     SIMPLEX
LAMBDA      1.8        0.5        5.1        2.4        10.2       7.8        3.4        2.9        2.5        2.1        5.6
Q            1          1          1          2          2          2          1          1          2          -          1

K            1          1          2          2          1          1          2          2          2          2          2
SPARES       2          2          2          3          4          4          2          2          3          1          3
COVERAGE  .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999
5 YR REL  .9995643   .9999898   .9956533   .9997877   .9745065   .9907581   .9986017   .9991110   .9997525   .9977565   .9991946
                                                     PRODUCT 1 = .9606708             PRODUCT 2 = .9974670
                                                                  MSC(FM) = .9560877                          MSC(CP) = .9577436

K            1          1          2          2          1          1          2          2          2          2          2
SPARES       1          1          2          3          4          4          2          2          3          1          3
COVERAGE  .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999   .9999999
5 YR REL  .9942525   .9995308   .9956533   .9997877   .9745065   .9907581   .9986017   .9991110   .9997525   .9977565   .9991946
                                                     PRODUCT 1 = .9551271             PRODUCT 2 = .9974670
                                                                  MSC(FM) = .9505704                          MSC(CP) = .9522167

K            1          1          2          2          1          1          2          2          2          2          2
SPARES       2          2          3          3          5          5          2          2          3          1          3
COVERAGE  .9999999   .9999999   .9900000   .9999999   .9900000   .9999999   .9900000   .9900000   .9999999   .9999999   .9900000
5 YR REL  .9995643   .9999898   .9970932   .9997877   .9789315   .9969080   .9971003   .9978280   .9997525   .9977565   .9966232
                                                     PRODUCT 1 = .9724275             PRODUCT 2 = .9946884
                                                                  MSC(FM) = .9650923                          MSC(CP) = .9669695

K            1          1          2          2          1          1          2          2          2          2          2
SPARES       2          2          3          3          4          5          2          2          3          1          3
COVERAGE  .9999999   .9999999   .9900000   .9999999   .9900000   .9999999   .9900000   .9900000   .9999999   .9999999   .9900000
5 YR REL  .9995643   .9999898   .9970932   .9997877   .9647535   .9969080   .9971003   .9978280   .9997525   .9977565   .9966232
                                                     PRODUCT 1 = .9583437             PRODUCT 2 = .9946884
                                                                  MSC(FM) = .9511147                          MSC(CP) = .9529648

K            1          1          2          2          1          1          2          2          2          2          2
SPARES       2          2          3          3          4          4          2          2          3          1          3
COVERAGE  .9999999   .9999999   .9900000   .9999999   .9900000   .9999999   .9900000   .9900000   .9999999   .9999999   .9900000
5 YR REL  .9995643   .9999898   .9970932   .9997877   .9647535   .9907581   .9971003   .9978280   .9997525   .9977565   .9966232
                                                     PRODUCT 1 = .9524317             PRODUCT 2 = .9946884
                                                                  MSC(FM) = .9452473                          MSC(CP) = .9470860
```

Figure 4—MSC reliability trade-off configurations

coverage shown in the figure. The next to the last configuration of the figure is the recommended one since the last one shows that decreasing the number of RWM spares to four drops both the MSC(FM) and MSC(CP) reliabilities below the required value.

Having illustrated that the required MSC five-year reliability can be met with either of two configurations it is necessary to choose between these. The reliabilities of the two are essentially equal. The main factors favoring the choice of the combined processor approach are: a smaller number of power switches, i.e., four vs. eleven; and a similar decrease in bus connections. Therefore the design using the combined processor approach was chosen as the baseline. This design and the appropriate sparing are summarized in Table I.

ACKNOWLEDGMENTS

The authors wish to acknowledge the support and assistance given by Capts. W. E. Kuehner, L. A. Fry, and W. E. Koss of the Air Force Space and Missile Systems Organization, and Drs. H. Hecht and W. A. Sturm, and Messrs. I. K. Egashira and D. J. Theis of The Aerospace Corporation. Appreciation is also extended to Mr. L. R. Murphy and Dr. R. L. Fulton for the many stimulating discussions.

REFERENCES

1 R A SHORT
The attainment of reliable digital systems through the use of redundancy—A survey
IEEE Computer Group News Vol 2 No 2 1968

2 A AVIŽIENIS G C GILLEY F P MATHUR
D A RENNELS J A ROHR D K RUBIN
The STAR (Self-Testing-And-Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design
IEEE Trans. on Computers Vol C-20 No 11 1971

3 W C CARTER et al
Design techniques for modular architecture for reliable computer systems
Report No 70-208-0002 IBM Space Systems Center
Huntsville, Alabama 1970

4 A L HOPKINS JR
A fault-tolerant information processing concept for space vehicles
IEEE Trans. on Computers Vol C-20 No 11 1971

5 A AVIŽIENIS
Arithmetic error codes: Cost and effectiveness studies for application in digital system design
IEEE Transactions on Computer Vol C-20 No 11 1971

6 I S REED D E BRIMLEY
On increasing the operating life of unattended machines
RM-3338-PR The RAND Corporation 1962

7 W G TEES
Predicting failure rates of yield-enhanced LSI
Computer Design February 1971

8 D M AARON M F ADAM
MOS reliability prediction model
Ninth Reliability and Maintainability Conference 1970

9 R L CUNNINGHAM
High reliability beam-lead devices
WESCON Session 20 1971

10 C M RYERSON et al
RADC reliability notebook volume II
RADC-TR-67-108 September 1967

11 F P MATHUR
Reliability estimation procedures and CARE: The computer-aid reliability estimation program
Jet Propulsion Laboratories Quarter Technical Review Vol 1 1971

MOS LSI minicomputer comes of age

by G. W. SCHULTZ and R. M. HOLT

American Micro-systems, Inc.
Santa Clara, California

INTRODUCTION

At the turn of the decade, a number of development programs were in progress to achieve a cross-breeding of the computer and semiconductor technologies. Since then, we have seen the advent of LSI mainframe memories. Further advances in LSI technology have incited considerable interest in other areas of computer applications, and as 1972 ends the LSI minicomputer comes of age.

Interestingly enough, the activities that led to this development were not stimulated by mainframe manufacturers, but by minicomputer users and terminal manufacturers in particular. In the course of this development, the tradeoffs that are necessary to make a minicomputer design a practical candidate for MOS LSI were of major concern to computer architects and systems designers. The factors that must be considered to attain an optimum LSI minicomputer system that operates at TTL speeds form the bases of this paper.

The specifications that apply in the discussions are:

1. The machine must be capable of addressing 65K words or bytes of memory directly. The 16-bit address to memory must not be sent as a single byte transfer, even in 8-bit oriented machines.
2. Speed is a major factor in achieving processing powers that approach TTL minicomputer designs; therefore, every possible effort must be made to gain speed.
3. TTL requirements external to the LSI should be minimized for cost effectiveness.
4. Only those LSI processing technologies that are presently being used in low-cost mass production should be considered.
5. The system should be able to function as an 8- or a 16-bit machine and its performance should be equivalent to TTL minicomputers that are most commonly used.

LSI MINICOMPUTER ARCHITECTURAL TRADEOFFS

Microprogrammed vs. Conventional Control

The tradeoffs in implementing the control section for the LSI machine are bounded primarily by two criteria: (1) system partitioning and related pin limitations; and (2) efficiency of chip area utilization.

If the entire machine could be placed on a single chip the problem would be greatly simplified and only the second restriction would remain. The other alternative is to partition the registers and arithmetic logic unit (ALU) as byte slices and place the control on other chips. However, this approach introduces speed problems because of chip-to-chip transition delays of approximately 150 to 200ns. The transition problem can be solved by using pipelining techniques which mask the time spent in communicating control between chips so that the registers and ALU section can be operated autonomously.

In byte slice partitioning, pin limitation is a major factor in the control section design. Present packaging technology and cost considerations constrain the designer to 16-, 24- and 40-pin packages. The 16- or 24-pin configurations are undesirable because the registers and ALU would have to be partitioned into 4-bit slices. The resultant chip-to-chip transitions between sections of the ALU will greatly reduce machine speed. Assuming that a 40-lead package is used:

1. At least 6 pins are required for power sources and clocks on all the chips.
2. 16 pins are needed on the registers chip for 16-bit memory Bus Interface.
3. At least three ALU Lines (Carry In, Carry Out and Zero Detection) are needed to form 16-bit machines.

This leaves a maximum of 15 pins for controlling the registers and ALU. This pin limitation problem combined with the fact that, for a given speed, Read-Only Memories (ROMs) are approximately 6 times more efficient in chip space usage than sequencers and associated random logic, leads the designer to seriously consider microprogramming techniques.

Aside from the basic LSI layout problems, other system tradeoffs heavily favor the microprogrammed approach. Much of the reasoning that applies to the design of TTL machines also applies to LSI designs.

Additional arguments for a microprogrammed approach are:

1. The need for emulating all or part of the instruction sets for existing TTL designs to minimize software development.
2. The need for applying the LSI machine to areas where the microlevel of control is either mandatory or desirable for meeting speed requirements. A good example is the CRT display terminal in which the microlevel of control is much more efficient, it may eliminate the need for core storage of macroinstructions and it allows the required speeds.
3. The desire for flexibility at the lowest logic level.
4. The short development spans which do not allow for complete re-layout of complex chips when minor errors are discovered during the design phase.

Instruction decoding

The instruction fetch and decode process poses a difficult speed problem. Particularly for the first phase of execution, it is desirable to load the Instruction Register (IR) with a new instruction and immediately relate the IR contents to a microinstruction ROM location. However, this ROM address mapping process typically requires a minimum of 300 ns as well as 300 to 400 ns to access the first microinstruction used in executing the macroinstructions. This is approximately equal to the time required to execute two microinstructions. This problem can be solved as shown in Figure 1 wherein the control section is designed to initiate the execution of one or more microinstructions after the instruction register is loaded, and as soon as the mapping array output is valid, it causes their completion to be inhibited or skipped. The inhibited microinstructions are used to fetch additional bytes in the process of forming a 16-bit memory address. For single-byte instructions, such as Clear Accumulator, the Fetch Address phase is skipped. For all other instructions the address is fetched and the instruction is decoded concurrently.

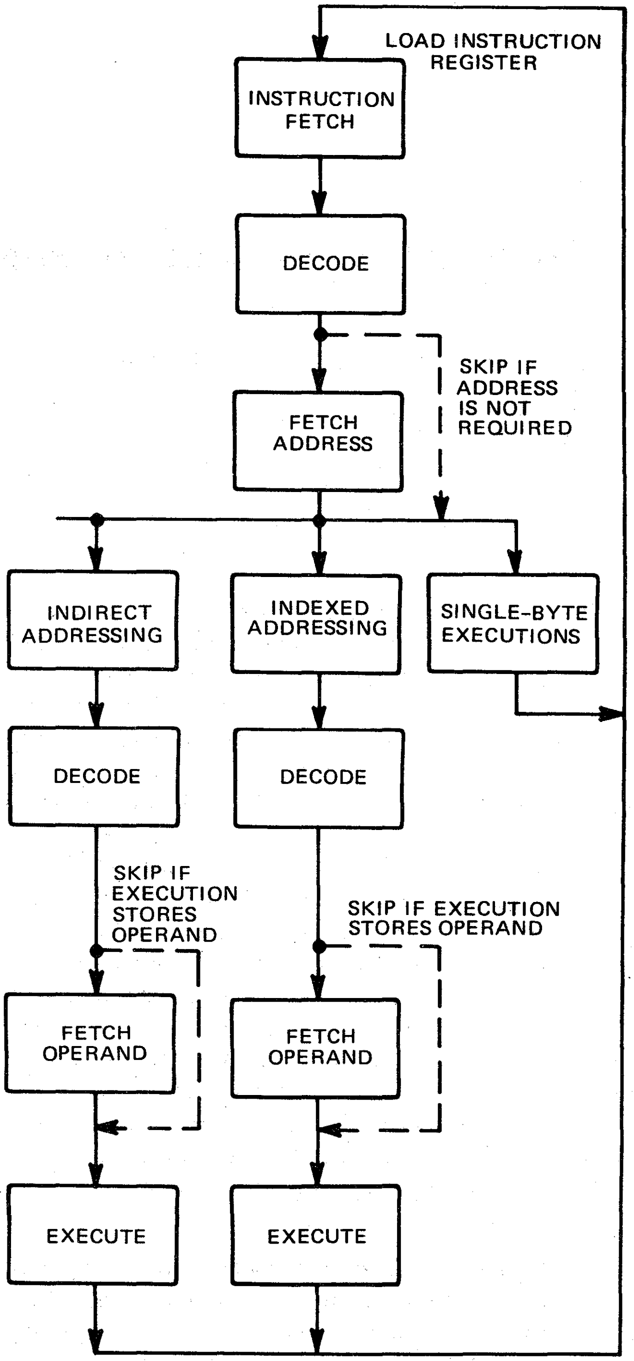


Figure 1—Instruction decoding

This same technique may be applied during the execution of memory reference instructions to differentiate between executions which store or fetch operands. In this case it is assumed that the execution phase will fetch an operand; if it is to store, the fetch operand microinstructions are inhibited or skipped.

Thus the Instruction Decode ROM provides ROM Addresses to the ROM Address Register (RAR) for each of the major execution phases:

1. Operand Address Preparation which includes

indirect, indexed and/or autoindexed addressing computation. More than one decode may be required for complex addressing schemes.
2. Execution of the instruction with the operand after address preparation or immediate execution for single-byte type instructions.
3. Occasional fetching and decoding of additional bytes or fields of the instruction for a third phase.

Fortunately, speed is not a problem for the second and third phases because the instruction is in the IR ahead of time.

Microinstruction formats

The speed of the LSI machine is determined primarily by the speed at which

1. Arithmetic operations can be performed, viz., the speed at which the operands can be accessed from two registers, added and returned into one of the original sources or a new register.
2. The machine can change control states. In a microprogrammed machine the ROM access speed and the microprogrammed branching process become determining factors.

The microprogrammed MOS LSI machine is optimized when the ROM access time equals the speed of arithmetics. If the machine is only used to interpret and execute macroinstructions, branching is not a major consideration, inasmuch as branching will seldom occur in the microprogramming process. If the machine is used mostly as a controller, the speed of branching will be an important factor. These factors influence the choice of microprogramming formats and field assignments.

In constructing the appropriate microinstruction formats the designer should take advantage of pipelining techniques which can be implemented at a very low cost with MOS LSI due to its dynamic storage node characteristics. This alone makes only a few TTL designs good candidates for LSI. Secondly, in an LSI machine the registers would most likely be implemented as two-port Random Access Memory (RAM) cells and must be accessed by two source register Address fields in much the same manner as any other semiconductor memory. While the two operands are being accessed the registers and ALU section need not know how to control the ALU or the destination of the results. Therefore, virtually no speed will be lost by communicating portions of the microinstruction format in a sequential manner as long as they are available in decoded form when required. This would suggest that machine time be an additional constraint in the construction of fields. It should be noted that if the system is pipelined extensively, two new operands must be accessed before the result of the last two operands is written into a destination register. Therefore it will be necessary to provide pipeline storage for the execution of microinstruction destination fields received by the registers and ALU section.

Presently, a single high speed (300ns access) microprogrammed MOS ROM contains as many as 12,000 bits, along with associated ROM addressing registers and control logic. Assuming that a 24-bit microinstruction format is used, 512 words are available which is adequate to implement a powerful minicomputer if the microaddress sequencing is optimized to reduce repetitive coding. One approach is the microsubroutine technique. Another equally powerful technique is to combine the decode operation and the CoCall. The decode operation retains the RAR + 1 in the ROM Return Address Register (RRAR) before the RAR is loaded with the Decode Address. The CoCall causes the RAR to be swapped with the RRAR (RAR + 1→ RRAR). The decode acts as a call to the microroutine and the CoCall acts as a return. CoCalls can also be used for branching between two routines.

Input/output interface

The designer should recognize that the MOS chip will drive only one TTL load efficiently and that power dissipation can be a major problem. Therefore it is necessary to consider an I/O buffering scheme which would provide the lightest possible load to the MOS chip in order to minimize speed and power dissipation. This can be accomplished by several techniques.

One interface technique is shown in Figure 2. When the MOS circuit drives data out the Drive/Receive control line is such that the receiver is in the tristate condition while the TTL buffer is driving the I/O Bus. When the MOS circuit is in the receiving mode the input data is driven sufficiently by the quad Driver/Receiver to overcome the data that are still being driven out from the output buffer and the MOS I/O chip need not know when the I/O Bus is driving or receiving.

The second technique, shown in Figure 3, requires separate Drive and Receive control lines, but it eliminates the need for the MOS output driver to be overcome by the data driven in. With this technique the output data is latched onto the I/O Bus, thereby lowering power dissipation on the chip. In smaller machines this could possibly eliminate the need for an I/O Buffer.

Of the two techniques the first requires fewer pads

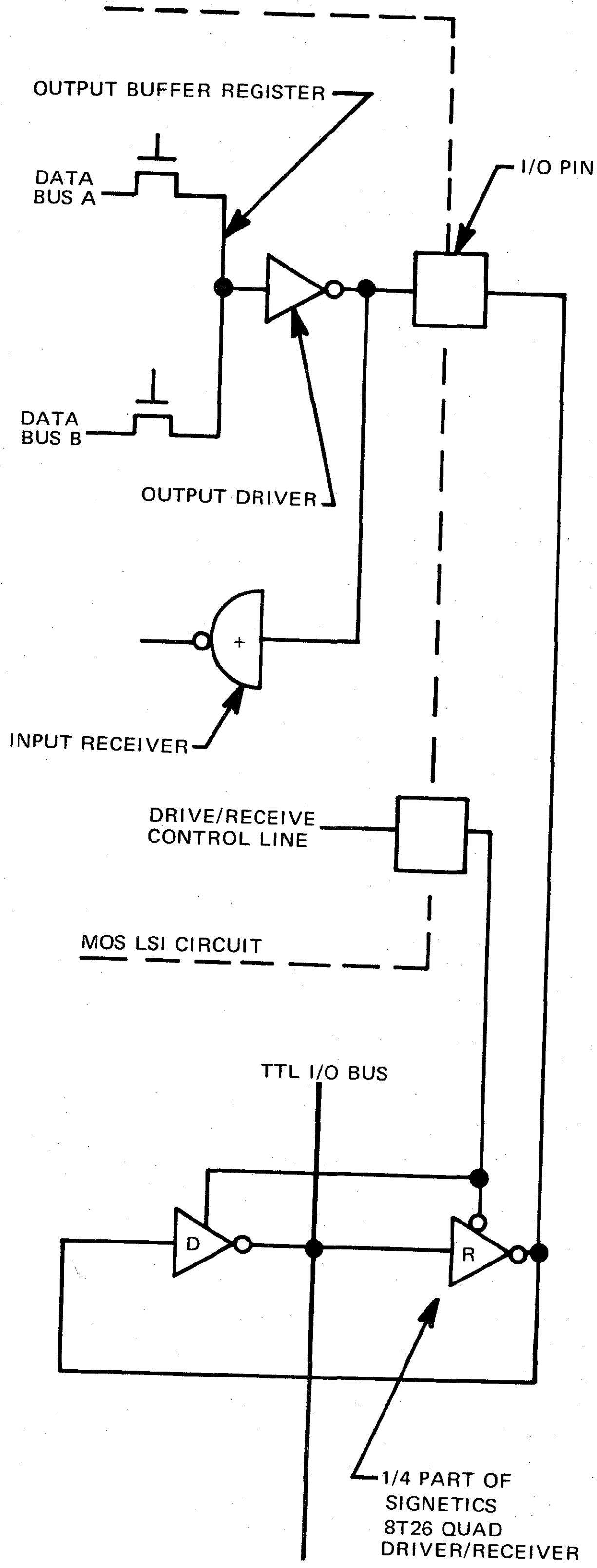


Figure 2—I/O interface with one control line

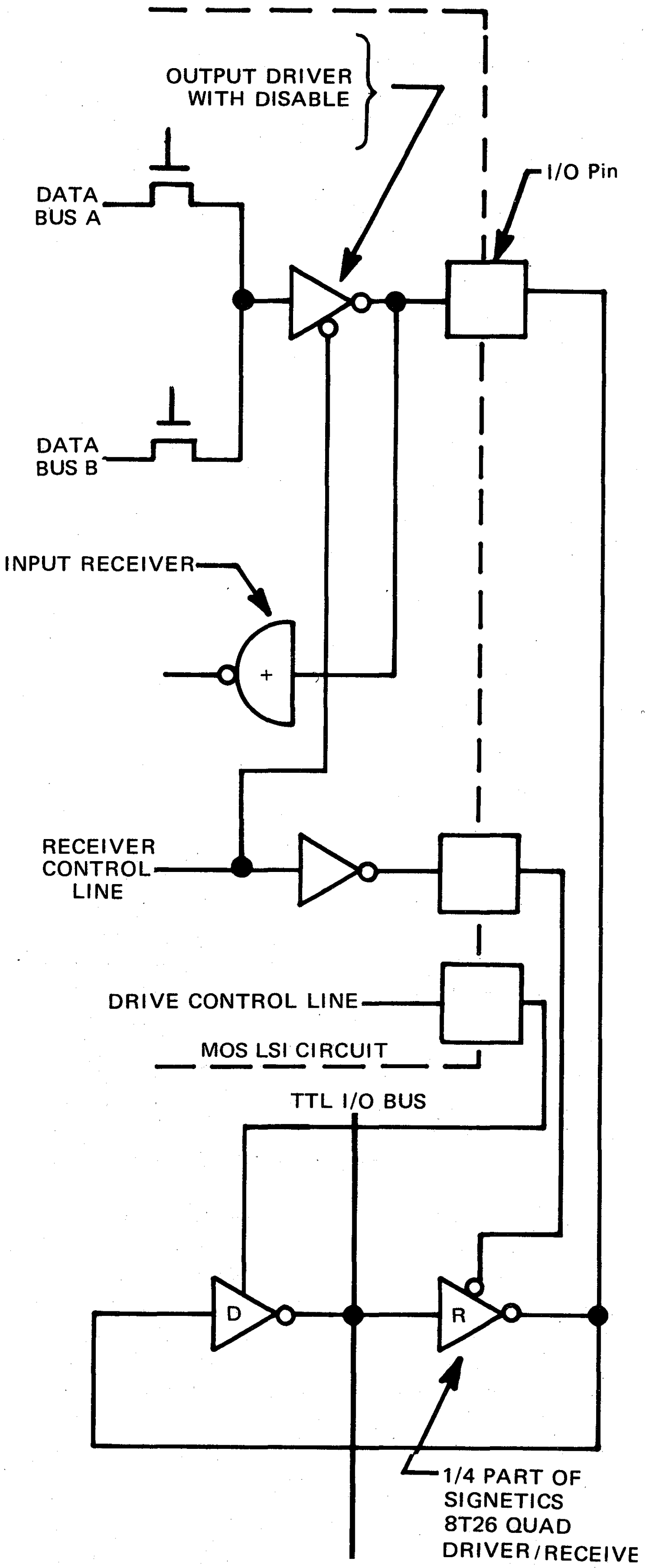


Figure 3—I/O interface with two control lines

and external interconnect, while the second dissipates less power on the chip.

Stack organization

The internal organization of a single address computer is such that both programming and running time are wasted for the storage of immediate results in the sequence of computation. This problem is amplified in the 8-bit byte machine because the full address must be fetched with the two memory accesses in order to transfer the operand. Moreover, memory products that are properly matched with these low-cost processors are likely to be slow. The two most efficient structures that can be implemented on LSI chips are ROMs and RAMs. Therefore it is desirable to consider an architecture that consists of a larger number of registers than are used in TTL designs; these would be used for temporary storage of operands.

Furthermore, it is highly desirable to have a push-down stack on the chip which would allow fast subroutine calls and interrupts which can then execute at microinstruction speeds. This feature is one of the most important factors in achieving the speeds equivalent to TTL designs. A single set of registers can be used either as a push-down stack or as a set of general file registers if the design incorporates a stack pointer counter which can be loaded to randomly address the registers within the set.

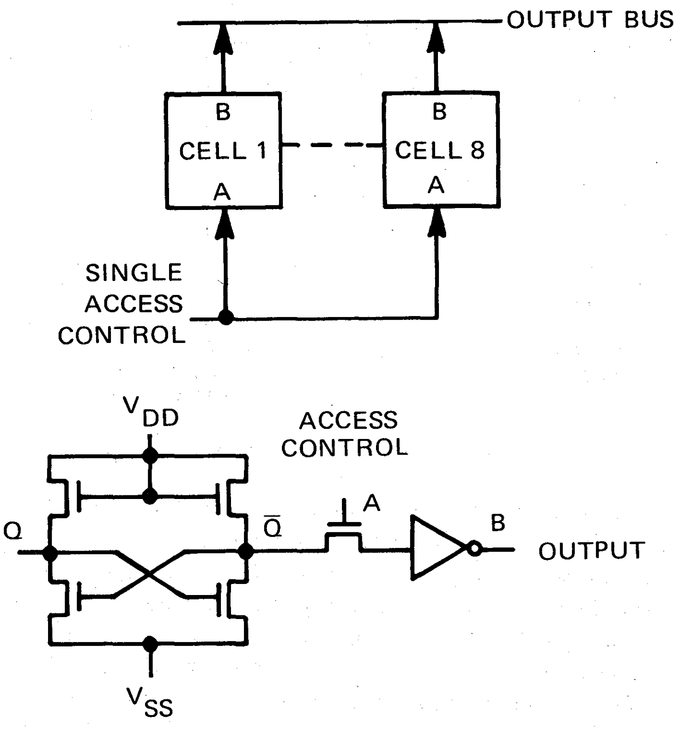


Figure 4—Single-port ram structure with static cell (input data and control not shown)

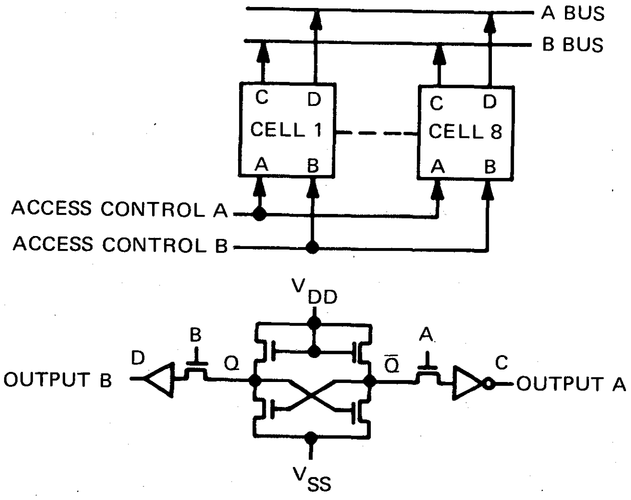


Figure 5—Dual-port ram structure with static cell (input data and control not shown)

Register implementation

The use of the RAM structure as a means of register implementation presents several interesting tradeoff considerations, such as single-versus dual-port structures, static versus dynamic cells and speed versus power.

In the single-port configuration shown in Figure 4 the general registers become one of the operands (A) and the accumulator or possibly the Program Counter becomes the other (B).

The dual-port structure shown in Figure 5 allows any of two general registers to be accessed simultaneously. In the dual-port configuration each RAM cell requires an additional access control line as well as output data line. Both of these lines can increase the size of each cell by 20 percent. The overall RAM structure including the additional outputs will increase by 30 percent. Along with the increase in the areas of the RAM structure, one must also consider the increase in interconnections for the additional eight output lines. The size tradeoffs for a single- and dual-port approach for ten general byte registers are:

	Single Port (mil^2)	Dual Port (mil^2)
Cell	18	22
RAM Core	1440	1760
Output Drivers	150	300
	1590	2060

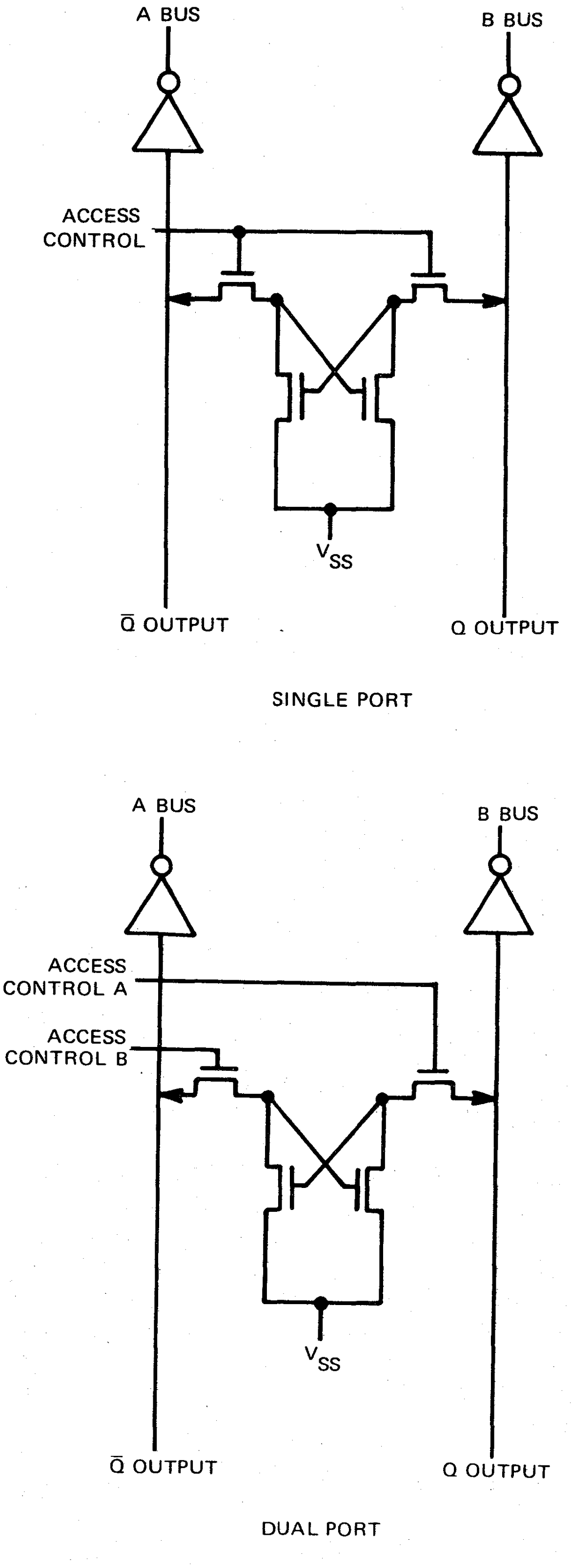


Figure 6—Refined ram static cell (inputs not shown)

It should be noted that the size of RAM cells have been decreasing constantly as new processing technologies have become available. Whereas an area of 18 mils^2 is practical today, it is very likely that the ion implant and silicon gate technologies will produce static RAM cells that are less than 10 mils^2.

The design process for register configurations must include consideration of the static and dynamic RAM cells. The static cell is the larger of the two and draws enormous power if the number of cells is large. This problem can be eliminated by relying on the capacitance at the Q, $\bar{Q}$ nodes to store information and by pulsing the load devices periodically. A refinement of this technique uses the Q, $\bar{Q}$ output data lines as sources of current to Q, $\bar{Q}$ nodes. This eliminates two devices from the cell structure but places a restriction on the size of the devices used to implement the cell. The single- and dual-port static cell structures are shown in Figure 6.

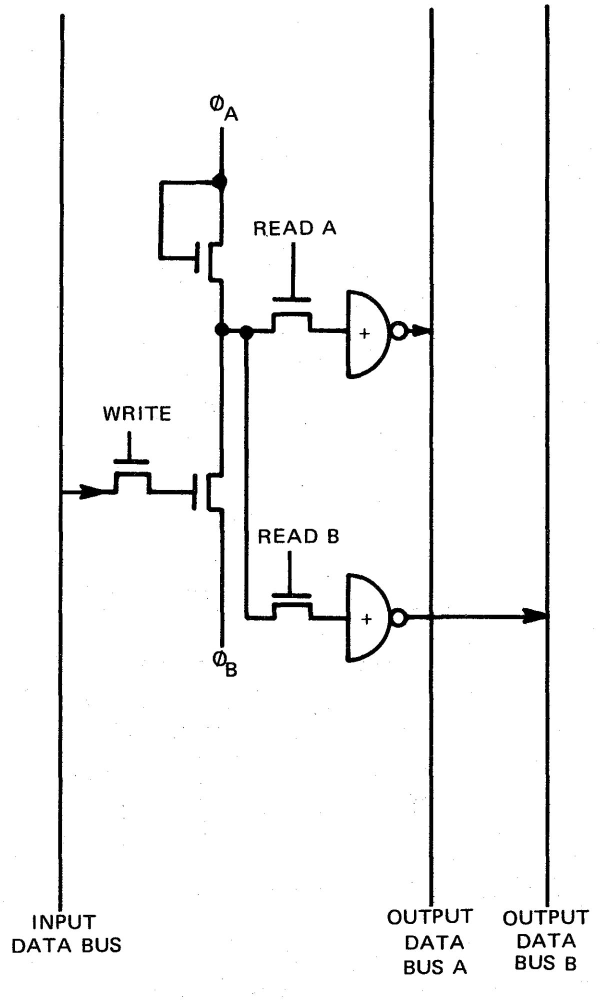
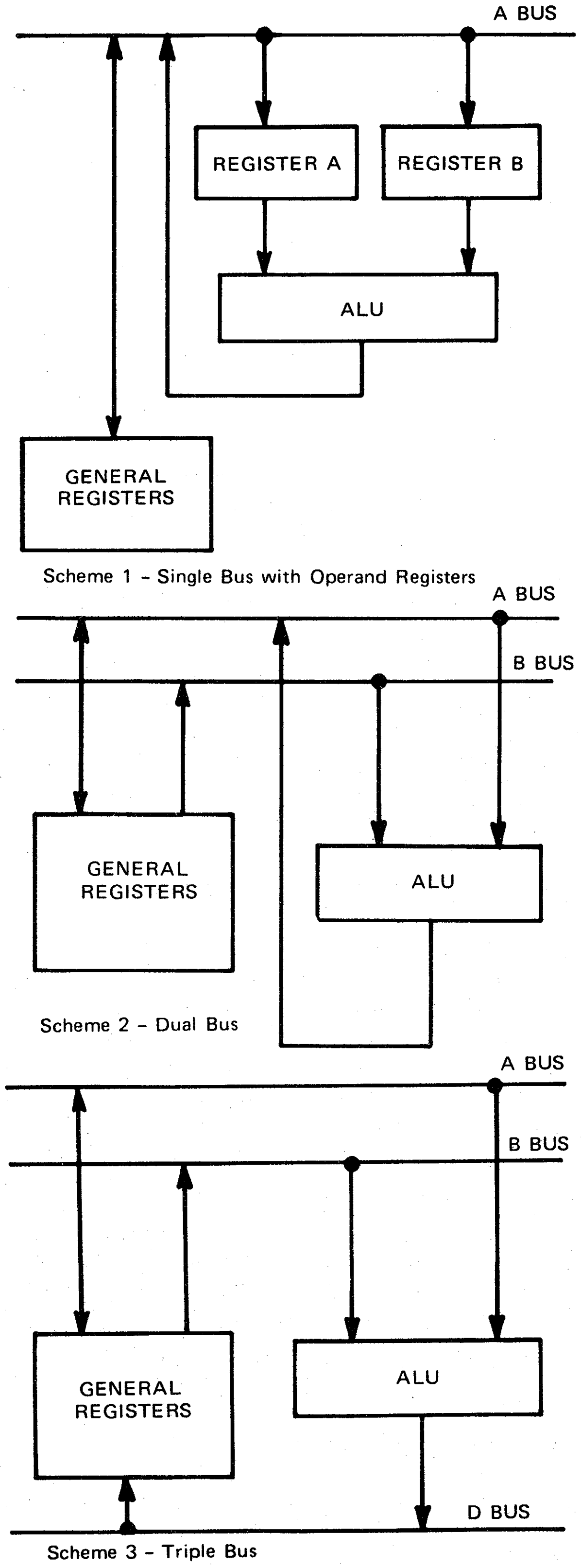


Figure 7—Dual-port dynamic ram cell



Scheme 1 - Single Bus with Operand Registers

Scheme 2 - Dual Bus

Scheme 3 - Triple Bus

Figure 8—Data bussing schemes

One advantage of the static cell is that it need not be refreshed.

The dynamic cell, shown in Figure 7, can be quite small due to its capacitive storage and it requires extremely low power. However, the dynamic cell must be refreshed, either with hardware or software. In either case, the processing stream must be interrupted periodically to refresh the data. Typically a dynamic cell must be refreshed at least once every millisecond (approximately once each 50μs at 125°C). For an MOS processor with 32 dynamic registers and a machine cycle of 1 μs, approximately 3 percent of the machine time would be spent in refreshing. Of course, as the cycle time increases this percentage would decrease, possibly to the point of insignificance. The system designer must determine whether this refresh time is worth the added complexity in control and timing logic versus the added chip area and power of a static cell. For an MOS LSI system with high computing performance requirements the dynamic RAM would not be a suitable choice. On the other hand, in systems where high performance is not the primary consideration, or the number of cells is larger, the dynamic MOS cell structure should certainly be considered.

Data bussing

The best criterion on bussing is to use the fewest number possible without sacrificing machine cycle speed. In some cases this will require that data be stored until the bus is available. Figure 8 shows three schemes that can be used to bus data around a chip. Scheme 1 uses a single bus which has the advantage of minimum interconnection but suffers in overall machine speed. In this type of organization the two operand registers must have two machine cycles and possibly a third cycle to transfer the result back to the general registers. In this case, hardware is traded for time.

Scheme 2 uses two busses to transfer the operands. The dynamic storage inherent with MOS allows these busses to look like registers. While this scheme eliminates one machine cycle for accessing the second operand, the general registers must have a dual-port structure.

Scheme 3 requires the greatest chip area and uses three busses, two for operands and one for the result. Pipelining techniques can be used in this scheme in that the A and B busses store operands while the D bus is being used to store the result of the previous operation.

For the sake of comparing the bussing schemes, consider the bus lines to run along three sides of the chip. The two layouts in Figure 9 show the relative importance of a proper layout.

Table I shows a comparison of bus chip area for the three bussing schemes in Figure 8 and for the two layout methods shown in Figure 9. Compared in the table are 150 and 200 mil^2 chips. Note that the amount of chip area consumed by bussing can vary from 4 to 30 percent depending on which bussing technique is used.

MOS LSI 7200 PROCESSOR

General

The AMI 7200 Processor is a fully parallel, bus-organized system wherein a 16-bit Data Exchange Bus interconnects the I/O devices, memory modules and central processors so they communicate with each other asynchronously. Multiprocessor capability is incorporated in the design of the Data Exchange Bus. The CPU contains 45 static registers of which 32 may, under microprogram control, be used as a First In-Last Out (FILO push-down) stack or as a file of 32 general registers.

The basic 8-bit 7200 processor is constructed of three ion implant, P-channel MOS LSI devices: (1) Registers and ALU Device; (2) Microinstruction ROM; and (3) Microcontrol Device.

TABLE I—Comparison of bus area for bussing schemes and chip layout methods for an 8-bit machine

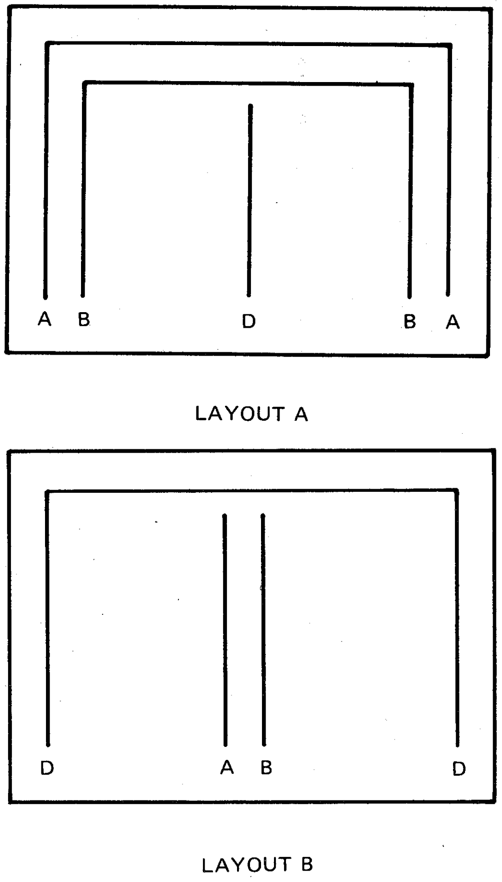
Layout A

	150 mil^2 Chip (Active Area)		200 mil^2 Chip (Active Area)	
Bus Scheme	Total Area of Busses* (mil^2)	Percent of Chip Area	Total Area of Busses (mil^2)	Percent of Chip Area
1	2880	12.8	3840	9.6
2	5760	25.6	7680	19.2
3	6720	30.0	8960	22.4

Layout B

	150 mil^2 Chip (Active Area)		200 mil^2 Chip (Active Area)	
Bus Scheme	Total Area of Busses* (mil^2)	Percent of Chip Area	Total Area of Busses (mil^2)	Percent of Chip Area
1	960	4.3	1280	3.2
2	1820	8.1	2560	6.4
3	4700	21.0	6400	16.0

* Areas are based on a line width of 0.4 mil with a spacing of 0.4 mil.



LAYOUT A

LAYOUT B

Figure 9—Chip bus layout methods

The registers and ALU device and the microinstruction device may be paralleled to form either 4- or 5-chip fully parallel 16-bit machines. The 7200 performs all of the functions commonly found in most minicomputers; the primary difference is the speed of execution which is approximately one-half to one-third of minicomputers. In a few cases the 7200 may be faster; for instance, the 32 byte stack allows for fast subroutine calls executed at microinstruction speeds rather than memory cycle speeds.

The basic machine architecture and partitioning are shown in Figure 10. The A, B and D busses which may also act as temporary storage registers are connected directly to the ALU. The ALU performs its operations in five time slots. While the ALU is operating on one set of operands, those for the next operation are being accessed from the general file registers or stack. After the next operands have been accessed the last result is written into the register as specified by microprogram control. The microinstruction timing is shown in Figure 11.

The Control section of the processor is implemented by microprogramming techniques. The microcontrol

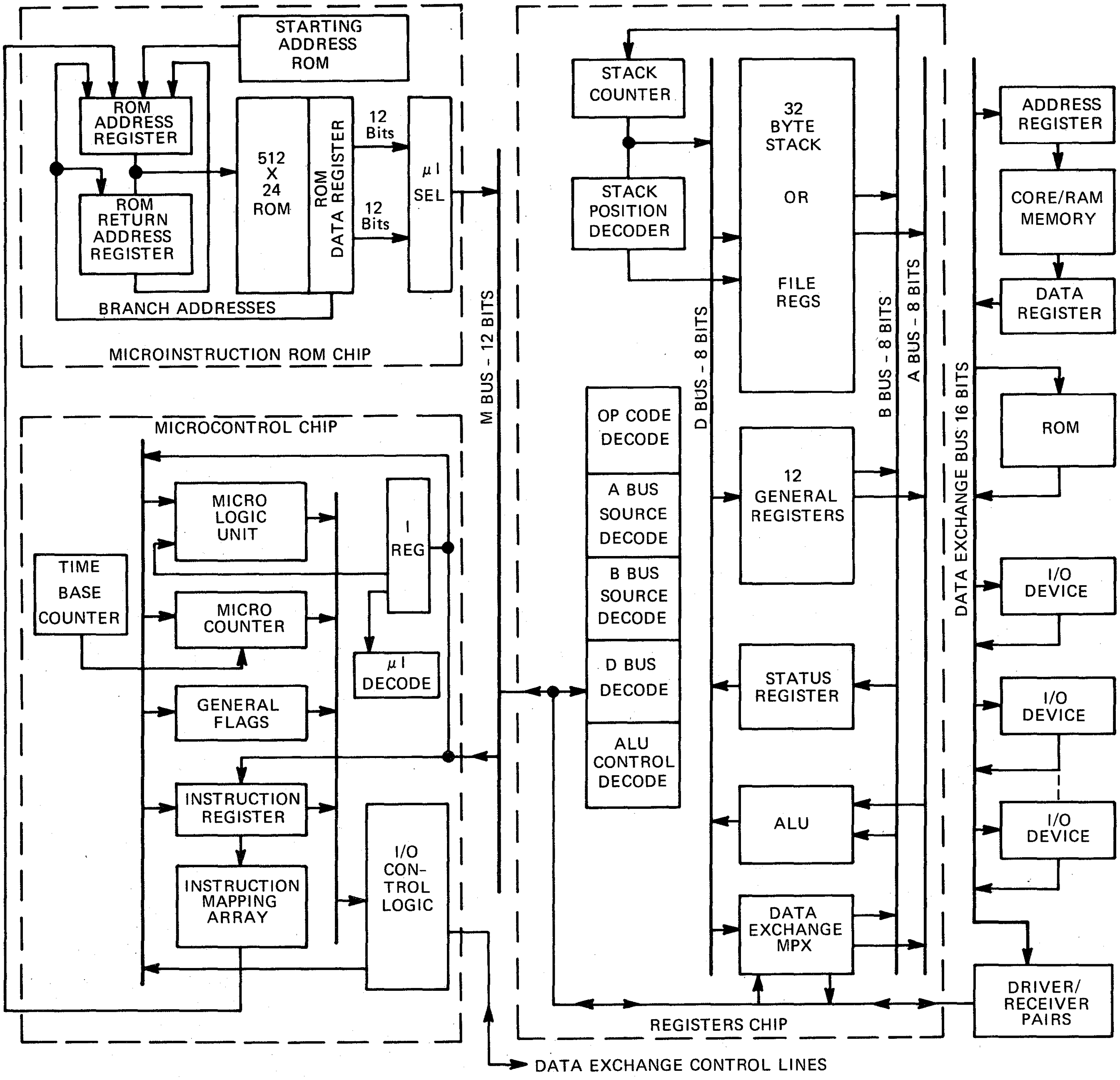


Figure 10—AMI 7200 processor

program is contained in the microinstruction ROM which has a maximum capacity of 512 words by 24 bits. The instruction decoding function is performed by the Instruction Mapping Array. The large microinstruction ROM and instruction mapping array allow for implementation of virtually any minicomputer instruction set. Twelve general purpose flags have been provided on the microcontrol chip to simplify the implementation of calculators and controllers. An interval timer with two time base ranges is also provided to aid in the design of control systems.

The logic was greatly reduced by using polynomial counters rather than binary counters. The RAR, RRAR, Time Base Counter and Stack Pointer are polynomial counters, while the microcounter is binary since the macrolevel of programming uses it.

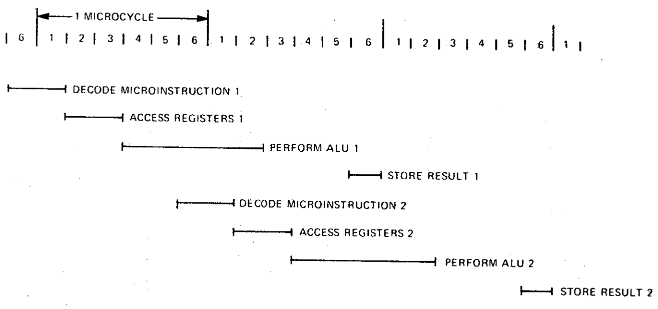


Figure 11—Timing chart for microinstruction pipelining in AMI 7200

7200 Microinstruction format

Four microinstruction formats for the 7200 processor, viz., Control, Literal, Test and Command, and Branch Address, are shown in Figure 12.

CONTROL FORMAT (Format 1)

4 Bits	4 Bits	4 Bits	4 Bits	8 Bits
OP Code	Bus A Source	Bus B Source	Bus D Destination	ALU Control
23 20	19 16	15 12	11 8	7 0

LITERAL FORMAT (Format 2)

4 Bits	8 Bits	4 Bits	8 Bits
OP Code	Literal	Bus D Destination	ALU Control
23 20	19 12	11 8	7 0

TEST AND COMMAND FORMAT (Format 3)

4 Bits	8 Bits	12 Bits
OP Code	Extended OP Code	Test or command
23 20	19 12	11 0

BRANCH ADDRESS FORMAT (Format 4)

3 Bits	9 Bits	1 Bit	1 Bit	1 Bit	9 Bits
OP Code	BRANCH ADDRESS 1	RC1	RCO		BRANCH ADDRESS 0
23 21	20 12	11	10	9	8 0

Figure 12—AMI 7200 instruction formats

Control format (format 1)

The Control Format is used to configure the registers in executing transfers and arithmetic operations. The Op Code field specifies Branching and Decoding Functions so that branching may be specified simultaneously with register control. The Bus A and Bus B source fields indicate one of 16 sources to be clocked onto these busses. Bus D destination stores the result of a register movement or arithmetic operation. The 32-byte stack is popped each time it appears as a source and pushed when it appears as a destination. The ALU control field specifies control of the arithmetic unit as well as shifting operations, I/O and status operations.

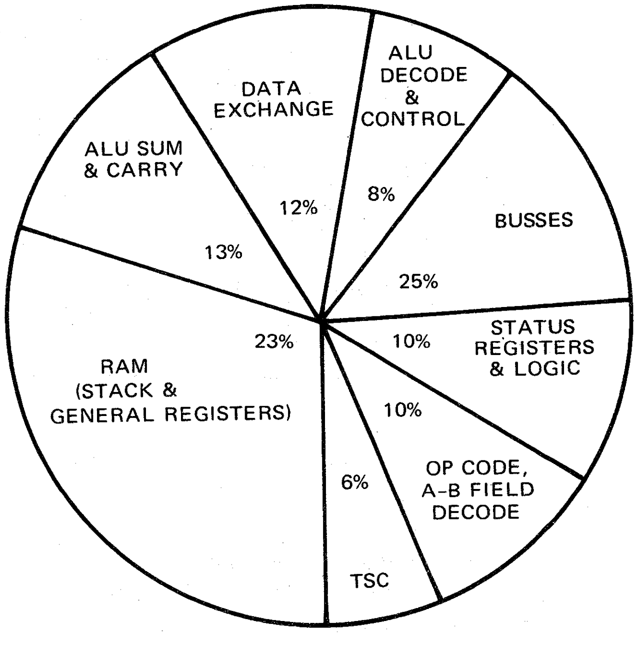
Literal format (format 2)

The Literal Format allows an 8-bit byte of any configuration to be clocked onto the B Bus and then used as an operand to the ALU as specified by the ALU destination field. The A Bus is loaded with the previous ALU result.

Test and command format (format 3)

The test and command format utilizes an extended Op Code field to further specify such operations as:

1. Setting and testing condition flags on the Register Chip.



TSC = TIME SLOT COUNTER

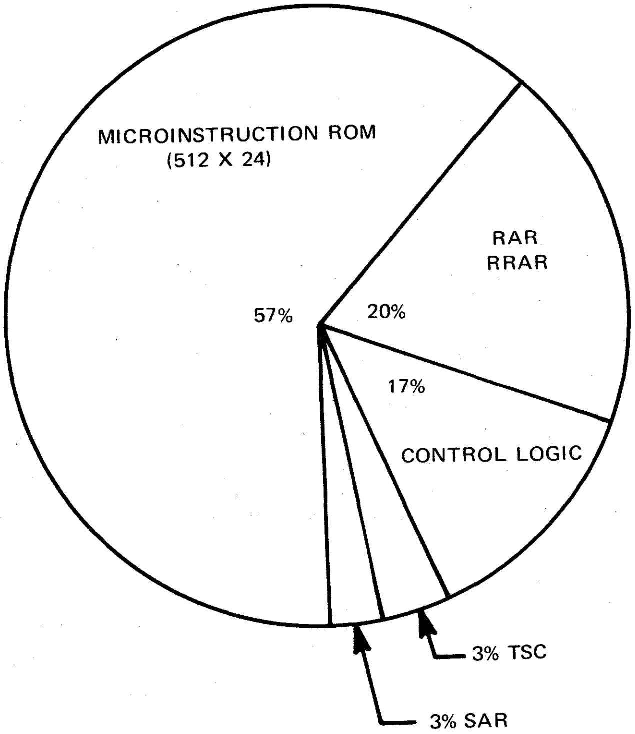
Figure 13—AMI 7200 registers and ALU chip

2. Branching-on condition and status flags.
3. Branching-on each bit of the instruction register.
4. Setting, resetting and testing general purpose flags.
5. Controlling the interval timer.

Branch address format (format 4)

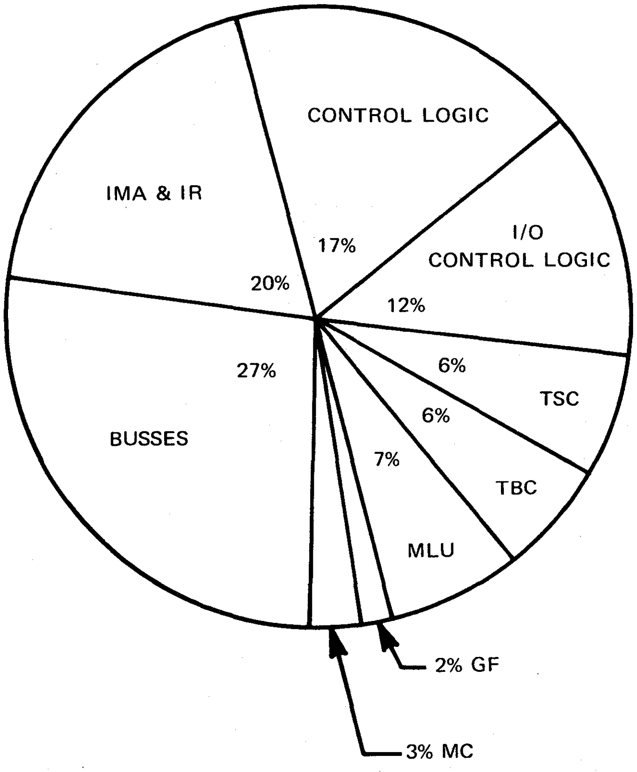
The branch address format contains two branch addresses that are used alone or with Format 3. When an unconditional branch parameter is specified, one branch address is used for a zero and the other for a one. Two bits of the format are used to specify control of the RAR and the RRAR as follows:

1. Load Branch Address 1 into RAR and leave RRAR undisturbed.
2. Load RAR + 1 into RRAR and Branch Address 1 into RAR.



RAR = ROM ADDRESS REGISTER
RRAR = ROM RETURN ADDRESS REGISTER
TSC = TIME SLOT COUNTER
SAR = STARTING ADDRESS ROM

Figure 14—AMI 7200 microinstruction rom chip



IMA = INSTRUCTION MAPPING ARRAY
IR = INSTRUCTION REGISTER
TSC = TIME SLOT COUNTER
TBC = TIME BASE COUNTER
MLU = MICROLOGIC UNIT
GF = GENERAL FLAGS
MC = MICROCOUNTER

Figure 15—AMI 7200 microcontrol chip

3. Load Branch Address 1 into RAR and Branch Address 2 into RRAR.
4. Load Branch Address 1 into RRAR.

Chip area allocations

The design results of the 7200 are included here to assist the system designer in planning new machines and effecting the tradeoffs discussed previously. The functional allocations for the Registers and ALU chip, the microinstruction ROM chip and the microcontrol chip are shown in Figures 13, 14 and 15 respectively. All chips are approximately 200 mils^2.

SUMMARY

The boundaries of computer performance that can be achieved through present MOS LSI processes approach those in corresponding TTL minicomputer designs when various architectural features and MOS design techniques are considered. MOS LSI technology makes it possible to incorporate various schemes in the computer architecture that were previously not economically available to the TTL designer. The AMI 7200 processor exemplifies a machine that has been optimized by using MOS LSI. The chip allocation data obtained from the 7200 will aid in planning new designs with architectures that would provide significant advantages in performance and economy.

Looking ahead, it is reasonable to expect that the new LSI processes that are now emerging, such as N-channel silicon gate, will make it possible to reduce chip sizes and increase operating speeds by a factor of two. Clearly, MOS LSI technology allows greater freedom in memory organization than other technologies in existence, and even wider applications will be achieved as system designers and computer architects become familiar with the characteristics of the new generation of MOS LSI devices.

ACKNOWLEDGMENT

The authors wish to acknowledge the invaluable assistance provided by the Circuit Designers at AMI and the Engineers of the Litton Advanced Retail Systems Division, particularly H. McFarland. We also extend our appreciation to the personnel of Comtec Data Systems, a subsidiary of AMI, for their assistance in the critique and preparation of the manuscript. Of course, the authors assume full responsibility for the accuracy and clarity of the content of this paper.

BIBLIOGRAPHY

L L BOYSEL J P MURPHY
Four-phase LSI logic offers new approach to comupter designers
Computer Design April 1970 pp 141-146

T C CHEN
Parallelism, pipelining and computer efficiency
Computer Design January 1971 p 69

R W COOK M J FLYNN
System design of a dynamic microprocessor
IEEE Transactions on Computers Vol 19 No 3 1970 pp 213-222

ENGINEERING STAFF OF AMERICAN MICRO-SYSTEMS, INC
MOS integrated circuits: theory, fabrication, design and systems application
Van Nostrand Rheinhold New York 1970

R S ENTNER
The advanced avionic digital computer system
Computer Design September 1970 pp 73-76

F FAGGIN M E HOFF
Standard parts and custom design merge in four-chip processor kit
Electronics April 24 1972 pp 112-116

R GRAHAM
The parallel and the pipeline computers
Datamation April 1970 p 68

R GRUNER L SELIGMAN J SUTTON
Standard LSI chips breed a fast new series of minicomputers
Electronics November 9 1970 pp 64-69

D C GUNDERSON
Some effects of advances in memory system technology on computer organization
IEEE Computer Group Conference Proceedings 1970 pp 7-11

M E HOFF
One chip CPU—computer or component
Proceedings of the Computer Systems Design Conference 1972 p 16

R M HOLT
MOS processor for the F14A CADC
Garrett AiResearch Corp Torrance California Technical Report No 71-7266 April 1971

F J LANGLEY
Small computer design using microprogramming and multifunction LSI arrays
Computer Design April 1970 pp 151-157

S R REDFIELD
A study in microprogrammed processors: a medium-sized microprogrammed processor
IEEE Transactions on Computers Vol 20 1971 pp 743-750

Control of the Rancho electric arm

by M. L. MOE and J. T. SCHWARTZ

University of Denver
Denver, Colorado

INTRODUCTION

In the past few years significant efforts have been made in the design of powered orthoses for upper extremities. These arm aids often consist of several linkages and many actuators making their control quite complex.

The critical problem to be discussed here is the development of a control system which will make it easy for the user to utilize the capabilities of the arm aid. The following are desirable properties of the control system:

1. Volitional control of the wrist of the arm aid over as wide a range of motion as possible.
2. Variable speeds to permit fast gross motion and slower speeds for fine positioning.
3. Direct control of wrist motion along natural trajectories using automatic coordination of individual joint motion.
4. Automatic features, such as keeping the hand level while drinking, available when the user desires.
5. A natural relationship between the source of control signals and the direction of motion desired.

While these properties determine the performance requirements of the control system, there are several additional factors which are of vital importance in determining patient acceptance of the complete system. These are:

1. Cosmesis—A disabled person has a desire to look as natural as possible. All equipment used should be designed to be as inconspicuous as possible.
2. Cost-Effectiveness—The additional function afforded the patient must be commensurate with the cost.
3. Reliability—The equipment should be designed to work for years without maintenance. If the equipment requires frequent maintenance it will tend to be discarded by the patient.
4. Ease of Application—It should be easy to put the equipment on the patient each time he wants to use it.
5. Simple Activation and Deactivation—It should be easy for the patient to turn the control system on and off with unique control signals. This will permit the patient to relax the muscles used to obtain control signals.

These requirements greatly increase the engineering effort required in the design of the arm aid, the control system, and the transducers used to obtain the required control signals. The objective of the research described here is the development of a complete arm aid system which has the specified control properties and also meets the additional requirements for patient acceptance.

The design objectives for the arm aid are somewhat different than those for most manipulators because of differences in operational environment and manipulative tasks. However, similar control strategies can often be used for both.

APPROACH

A block diagram of the complete control system under development is shown in Figure 1. Control sites capable of providing signals, $\underline{r}_i$, which can be smoothly controlled over a wide dynamic range, are essential to obtain suitable speed control. The control system is designed to be of assistance to quadriplegics, generally a a result of a high-level spinal cord injury. This places severe constraints on the location of control sites which are not present in controllers for most manipulator systems. A transducer which monitors eye motion by means of infrared reflection techniques is being developed to provide two of these signals since smooth control of eye

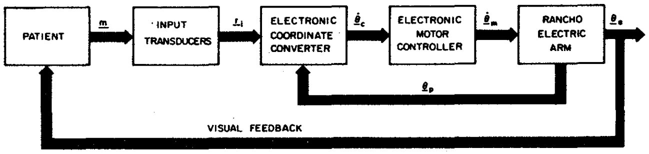


Figure 1—Block diagram of system

motion is available in even the most severely paralyzed patients. Additional control sites are chosen to utilize other residual function.

To simplify the control task an electronic coordinate converter is employed to compute the rate of motion for each joint of the arm aid when the desired direction and rate of speed of the hand is specified. The conscious effort required for control of the hand is minimized since the user can use the control transducers to specify speed and direction of motion of the hand in familiar coordinate systems rather than specifying anatomical joint motion. In order to achieve the desired coordinated motion, variable motor speed control is needed. Since the arm aid used, the Rancho electric arm,[1] has permanent magnet dc motors it was necessary to generate a variable duty cycle pulse train to obtain the necessary range of motor speeds.

OCULAR TRANSDUCER

Previous tests using EOG methods for monitoring eye motion for control purposes indicated that such control was feasible.[2,3,4] However, because the use of electrodes did not seem compatible with the requirements of patient acceptance, this method of monitoring eye motion was discarded in favor of a method using infrared reflection techniques. This new transducer, shown in Figure 2, uses infrared sources and detectors placed so that they do not interfere with normal lines of vision. Vertical motion is sensed by the source and two detectors above the right eye and horizontal motion is sensed by the sources and detectors on the sides of the glasses. By modulating the sources it was possible to design the transducer to operate over a wide range of ambient light conditions. A block diagram of the ocular transducer is shown in Figure 3 where the blocks containing S are sources and those containing a D are detectors. A study of infrared hazards to the eye indicate that the power levels used for the sources produces infrared radiation well below the threshold of damage to the eye.[5]

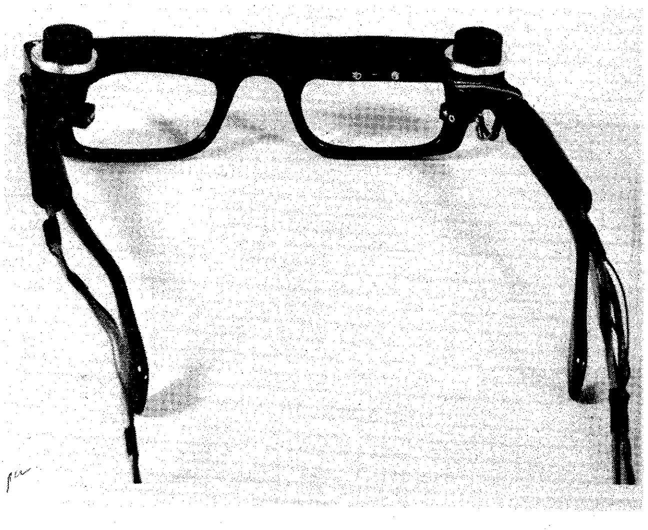


Figure 2—Infrared ocular transducer

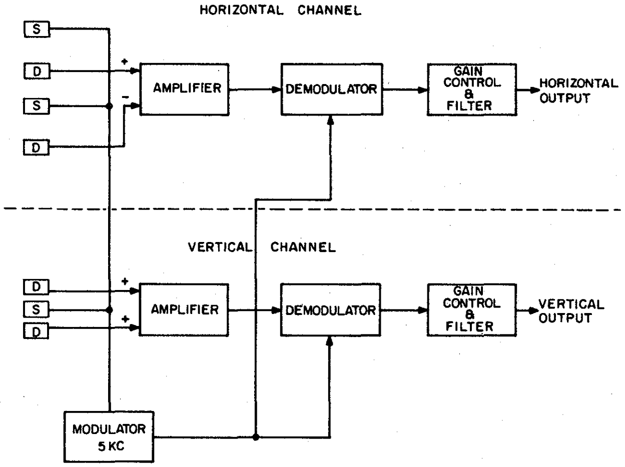


Figure 3—Block diagram of ocular transducer electronics

The control currently available from the ocular transducer is illustrated by the eye "writing" shown in Figure 4. This writing was obtained by attaching the vertical and horizontal outputs of the ocular transducer to the vertical and horizontal inputs of a storage oscilloscope.

It is important that the patient be able to turn off the control system quickly when he wants to look around or talk to someone. Two approaches to this problem are currently being investigated. The first method uses the occurrence of two blinks within a 175 to 350 millisecond period to disconnect the transducer. The other method uses the presence of a zero signal in both channels for a specified period, as the indication. Of course, similar methods can be used to reconnect the transducer.

ELECTRONIC COORDINATE CONVERTER

Many techniques can be used to perform the calculations necessary to convert the input signals specifying direction and speed into signals specifying individual joint rates. Since these calculations effectively transform signals from a spatial coordinate system into the coordinate system of the arm, the device performing the calculations will be called a coordinate converter. Earlier research used a mechanical coordinate converter based on spherical coordinates for the spatial coordinate system.[2,3,4] Zebo[6] has used a mechanical coordinate converter based on Cartesian coordinates. Because of slow operating speeds and lack of flexibility, these mechanical coordinate converters have given way to electronic coordinate converters. Many approaches to the coordinate conversion have appeared in the literature. Whitney[7,8] has proposed a coordinate system based upon the direction the hand is pointing. Lawrence and Lin[9] have proposed a statistical method for determining the elbow positions of an arm aid. Gavrilovic and Maric[10] have developed equations for keeping the hand direction colinear with wrist velocity. Singh[11] developed the coordinate transformation equations for spherical coordinates with the center of the coordinate system at the shoulder. This was later extended by Greeb[12] to allow the center of the spherical coordinate system to be displaced from the shoulder. Greeb also developed the equations for a Cartesian coordinate system. If the center of the spherical system is located at the mouth then a single signal can be used to bring the hand to the mouth. Thus, the spherical coordinate system would seem to be good for eating. However, the Cartesian system may be better for lapboard activities when motion parallel to the board is desired.

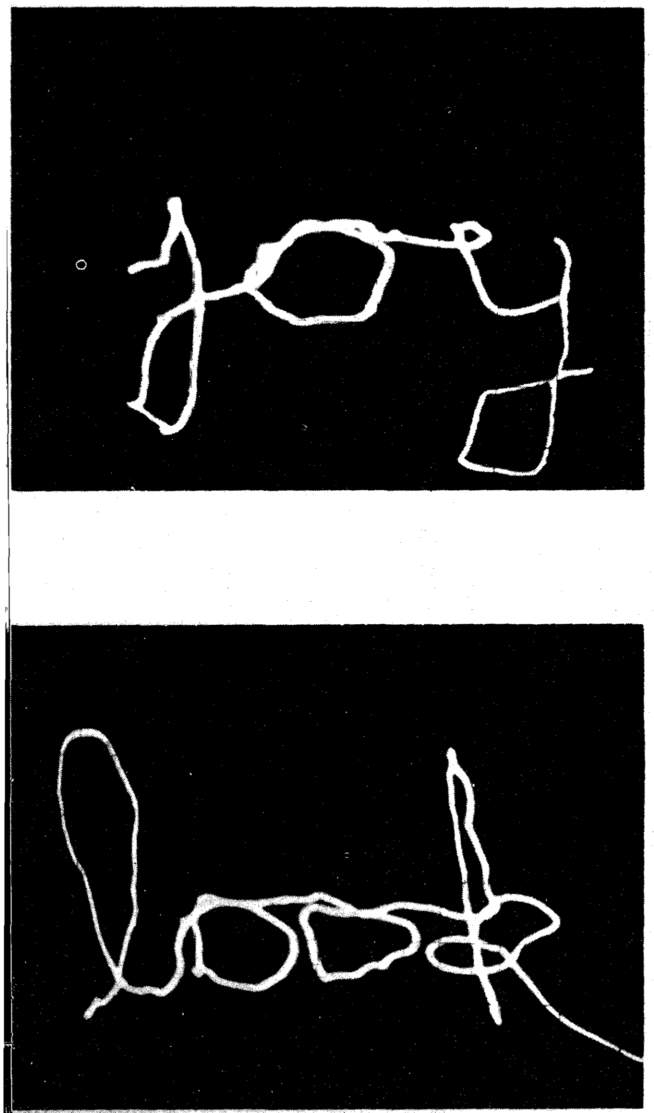


Figure 4—Eye "writing" using infrared ocular transducer

Any implementation of a coordinate converter must make specific choices to resolve three situations which arise. The first is a result of the two-bar linkage having four degrees-of-freedom when only three degrees-of-freedom are required to define a point in space. This extra degree-of-freedom makes the solution of the coordinate transformation equations non-unique. Each resolution of this ambiguity leads to a different control strategy. Some strategies proposed are:

1. Minimization of instantaneous weighted system kinetic energy[7]
2. Hold one joint, such as humeral rotation, fixed[12]
3. Use statistical procedures to determine whether to use high or low elbow position[9]

None of these strategies has yet received enough testing to determine which would be preferred by a user.

The second situation arises when one of the joints of the arm aid reaches a limit. In this case, the joint cannot move further so coordinated motion is destroyed if motion toward that limit is required. Some of the possible resolutions of this problem are:

1. Permit uncoordinated motion by stopping the joint at the limit but letting the other joints continue moving. This procedure can be unnerving to the user as it often appears to him that he has lost control.
2. Stop the arm aid when any joint reaches a limit and ignore all input commands which require motion against the limit. This often severely

limits the range of motion of the arm. Also, since the inputs are end-point commands and not joint commands, it is often difficult for the patient to determine a command that will permit motion away from the limit. The lack of a response to input commands which are illegal because they require motion into a limit is very discomforting to the user.

3. Control the arm to move in a direction which most closely approximates the desired direction. This would greatly complicate the control algorithm although it may be necessary to give the patient an adequate feel of control.

While the first two strategies have been used in preliminary tests, none has been used extensively enough to evaluate its effectiveness.

The third situation arises because of the singularities which must exist in the solution of the coordinate transformation equations because of the mechanical structure of the arm aid. Some strategies which can be used to avoid the singularities are:

1. Adjust the limits of the arm aid so it cannot be driven to a point where a singularity exists.
2. Modify the arm position data when close to a singularity so that the position data used by the equations do not result in a singular solution. Since the actual arm angles are then a few degrees from the angles used in the coordinate conversion process, a degradation in coordinated motion results. However, in general the change in performance is not noticeable to the user.
3. Adopt a different control algorithm in the vicinity of a singularity. Because of the unique position of the arm near a singularity a simple strategy such as temporarily going directly to joint angle control may be adequate.

In the system currently being tested, as shown in the block diagram of Figure 1, the coordinate converter computes the desired joint rates $\dot{\underline{\theta}}_c$, specified by the command signals, $\underline{r}_i$, and information on the current position of the arm, $\underline{\theta}_p$, as measured by potentiometers. Both Cartesian and spherical coordinate systems are being used by the coordinate converter for evaluation. At present, the extra degree-of-freedom problem is being resolved by letting the patient directly control humeral rotation so its rate is not computed by the coordinate converter.

The equations developed by Greeb[12] are those currently being used in the electronic coordinate converter. The origin of the reference coordinate system is located at the mouth, and oriented such that $+X$ is forward, $+Y$ is to the right, and $+Z$ is down. The constant vector, S_0, where

$$S_0=\begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$$

defines the location of the shoulder with respect to the origin.

The location of the wrist in space is defined by four joint angles as shown in Figure 5. The reference position is the arm extended to the front parallel with the X axis. From this reference position a positive rotation of ψ_1 moves the wrist to the right; a positive rotation of θ_1 raises the wrist and a positive rotation of φ_1 rotates the arm clockwise when viewed from the shoulder. Positive rotation of θ_2 produces elbow flexion bringing the wrist closer to the shoulder. The components of the

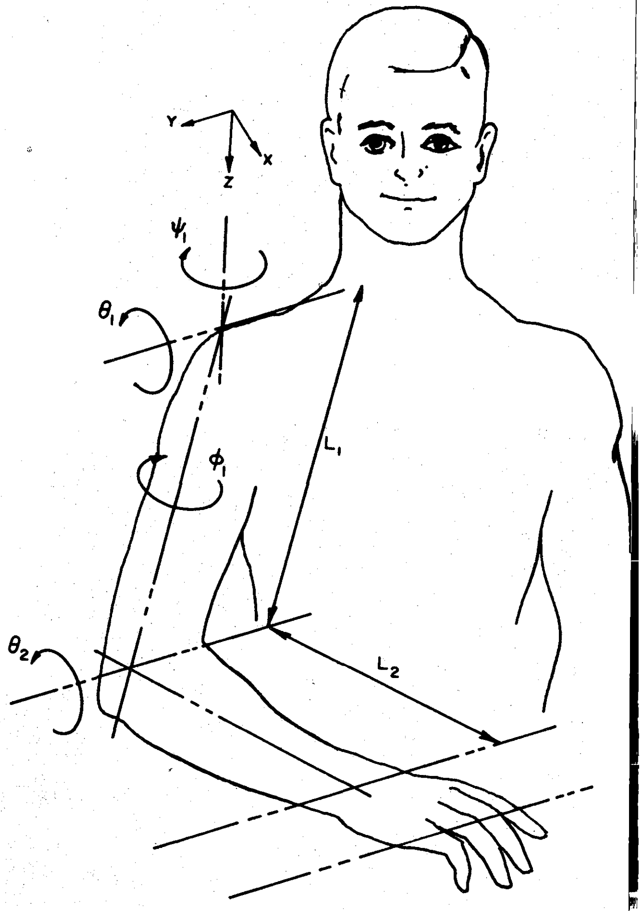


Figure 5—Angles defining wrist position in space

wrist position can be expressed as:

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix} + \begin{bmatrix} C\psi_1 C\theta_1 L_1 + (C\psi_1 C\theta_1 C\theta_2 \\ \quad - C\psi_1 S\theta_1 C\varphi_1 S\theta_2 - S\psi_1 S\varphi_1 S\theta_2) L_2 \\ S\psi_1 C\theta_1 L_1 + (S\psi_1 C\theta_1 C\theta_2 \\ \quad - S\psi_1 S\theta_1 C\varphi_1 S\theta_2 + C\psi_1 S\varphi_1 S\theta_2) L_2 \\ - S\theta_1 L_1 - (S\theta_1 C\theta_2 + C\theta_1 C\varphi_1 S\theta_2) L_2 \end{bmatrix} \tag{1}$$

where

L_1 = Length of upper arm

L_2 = Length of lower arm

and C and S are used to designate the Cosine and Sine, respectively.

In order to simplify the equations, the upper and lower arm will be assumed to be of the same length: $L = L_1 = L_2$. This is actually an advantage since it permits control of the palm of the hand rather than the wrist, although we will still call it the wrist.

The rate equations for the wrist can be written:

$$\begin{bmatrix} \dot{X}_w \\ \dot{Y}_w \\ \dot{Z}_w \end{bmatrix} = LA \begin{bmatrix} \dot{\psi}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} \tag{2}$$

where it is assumed φ_1 is held constant. Thus, φ_1 is not used in the coordinate conversion process but rather is placed under direct control of the patient. The components of A can be written as follows:

$$\left.\begin{aligned} a_{11} &= S\psi_1[S\theta_1 C\varphi_1 S\theta_2 - C\theta_1(1+C\theta_2)] - C\psi_1 S\varphi_1 S\theta_2 \\ a_{12} &= -C\psi_1[S\theta_1(1+C\theta_2) + C\theta_1 C\varphi_1 S\theta_2] \\ a_{13} &= -[C\psi_1(C\theta_1 S\theta_2 + S\theta_1 C\varphi_1 C\theta_2) + S\psi_1 S\varphi_1 C\theta_2] \\ a_{21} &= C\psi_1[C\theta_1(1+C\theta_2) - S\theta_1 C\varphi_1 S\theta_2] - S\psi_1 S\varphi_1 S\theta_2 \\ a_{22} &= -S\psi_1[S\theta_1(1+C\theta_2) + C\theta_1 C\varphi_1 S\theta_2] \\ a_{23} &= -[S\psi_1(C\theta_1 S\theta_2 + S\theta_1 C\varphi_1 C\theta_2) - C\psi_1 S\varphi_1 C\theta_2] \\ a_{31} &= 0 \\ a_{32} &= S\theta_1 C\varphi_1 S\theta_2 - C\theta_1(1+C\theta_2) \\ a_{33} &= S\theta_1 S\theta_2 - C\theta_1 C\varphi_1 C\theta_2 \end{aligned}\right\} \tag{3}$$

The desired joint speeds can then be computed from Equation (2) to obtain:

$$\begin{bmatrix} \dot{\psi}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = \frac{1}{L} A^{-1} \begin{bmatrix} \dot{X}_w \\ \dot{Y}_w \\ \dot{Z}_w \end{bmatrix} \tag{4}$$

As one would expect, the equations reveal that the joint rates are not dependent on the position of the origin of the coordinate system relative to the shoulder.

For the spherical coordinate system, the origin will again be assumed to be the mouth. The position of the wrist will be defined by the radial distance, R, the azimuth angle, θ, and the elevation angle, φ. The input commands will be Ṙ, $\dot{\theta}$, and $\dot{\varphi}$, and defined such that positive $\dot{R}$ moves the hand away from the mouth, a positive $\dot{\theta}$ moves the wrist to the right and a positive $\dot{\varphi}$ raises the wrist. The rate equations can be written:

$$G \begin{bmatrix} \dot{R} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = H \begin{bmatrix} \dot{\psi}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} \tag{5}$$

Before defining the elements of G and H, we first define:

$$\left.\begin{aligned} r &= (X_0^2 + Y_0^2)^{1/2} \\ \alpha &= \mathrm{Tan}^{-1}\,(Y_0/-X_0) \\ T &= L(C\theta_1 + C\theta_1 C\theta_2 - S\theta_1 C\varphi_1 S\theta_2) - rC(\alpha+\psi_1) \\ B &= LS\varphi_1 S\theta_2 + rS(\alpha+\psi_1) \\ z_w &= Z_0 - L(S\theta_1 + S\theta_1 C\theta_2 + C\theta_1 C\varphi_1 S\theta_2) \\ R &= (T^2 + B^2 + z_w^2)^{1/2} \end{aligned}\right\} \tag{6}$$

Then the elements of G may be written:

$$\left.\begin{aligned} g_{11} &= (T^2+B^2)/R \\ g_{12} &= 0 \\ g_{13} &= z_w(T^2+B^2)^{1/2} \\ g_{21} &= 0 \\ g_{22} &= T^2+B^2 \\ g_{31} &= R \\ g_{32} &= g_{33} = 0 \end{aligned}\right\} \tag{7}$$

and the elements of H may be written:

$$\left.\begin{aligned} h_{11} &= rTS(\alpha+\psi_1) + rBC(\alpha+\psi_1) \\ h_{12} &= -LT(S\theta_1 + S\theta_1 C\theta_2 + C\theta_1 C\varphi_1 S\theta_2) \\ h_{13} &= -LT(C\theta_1 S\theta_2 + S\theta_1 C\varphi_1 C\theta_2) - LBS\varphi_1 C\theta_2 \\ h_{21} &= T^2 + B^2 - rBS(\alpha+\psi_1) + rTC(\alpha+\psi_1) \\ h_{22} &= LB(S\theta_1 + S\theta_1 C\theta_2 + C\theta_1 C\varphi_1 S\theta_2) \\ h_{23} &= LB(C\theta_1 S\theta_2 + S\theta_1 C\varphi_1 C\theta_2) + LTS\varphi_1 C\theta_2 \\ h_{31} &= h_{11} \\ h_{32} &= -LT(S\theta_1 + S\theta_1 C\theta_2 + C\theta_1 C\varphi_1 S\theta_2) \\ &\quad + Lz_w(S\theta_1 C\varphi_1 S\theta_2 - C\theta_1 - C\theta_1 C\theta_2) \\ h_{33} &= -LT(C\theta_1 S\theta_2 + S\theta_1 C\varphi_1 C\theta_2) + LBS\varphi_1 C\theta_2 \\ &\quad + Lz_w(S\theta_1 S\theta_2 - C\theta_1 C\varphi_1 C\theta_2) \end{aligned}\right\} \tag{8}$$

The desired joint speeds can then be computed from Equation (4):

$$\begin{bmatrix} \dot{\psi}_1 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = H^{-1} G \begin{bmatrix} \dot{R} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{9}$$

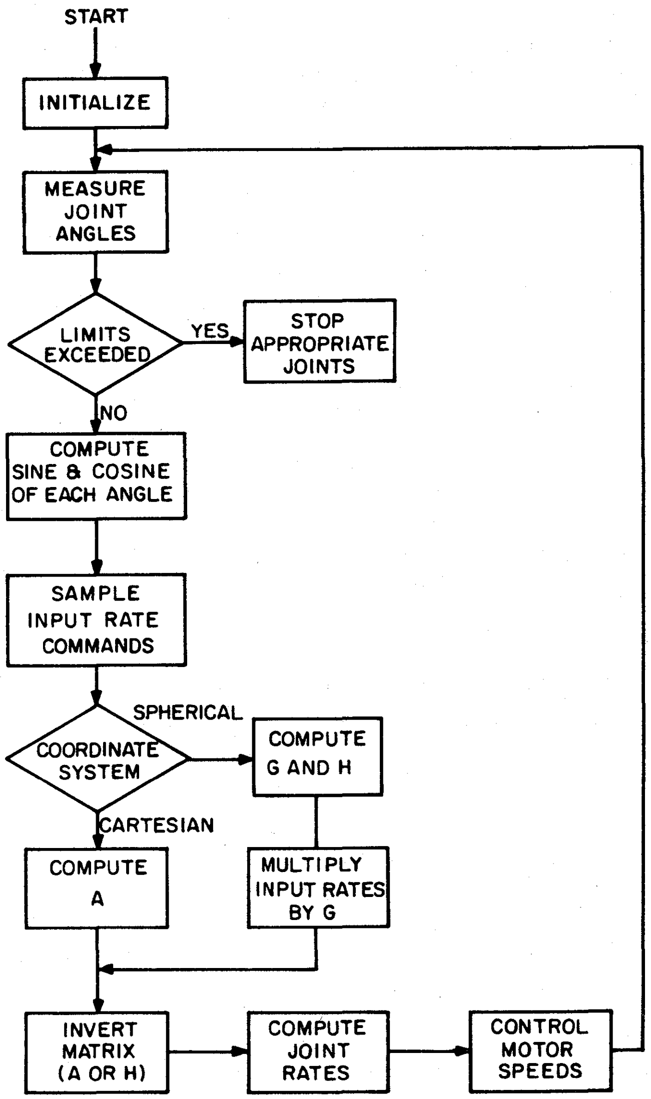


Figure 6—Flow diagram of computer program

It can be seen that the coefficients of G and H are somewhat more complex to calculate than those of A. Also, since the spherical coordinate system uses the mouth-to-shoulder offset, S_0, these measurements of the patient must be available in addition to arm length.

The prototype electronic coordinate converter was implemented using a Data General Nova 1200 minicomputer with a $4K$ core. A flow diagram of the program is shown in Figure 6. A front panel switch is used to determine whether the Cartesian or spherical coordinate system is used. This switch can be changed during program operation and allows the patient to make a quick comparison between the Cartesian and spherical coordinate systems for any task.

Since the computer used does not have the hardware multiply option, the number of multiplications required is an important factor when using fixed point calculations. The implementation of the Cartesian coordinate system requires 51 multiplications, whereas 68 are required for the spherical coordinate system. If direct substitution is used instead of matrix inversion for the Cartesian coordinate system, the number of multiplications could be reduced to 36.

Simple high-speed floating-point subroutines were developed for the Nova 1200 to simplify programming during the initial development phases. In this system the mantissa is stored in one 16-bit word and the exponent in the following 16-bit word. Although this does not make most effective use of memory, it permits much faster floating-point operations than the standard routines.

The execution time for each floating-point routine is given in Table I. For addition and subtraction, shifts are required to make the exponents equal, but no shifts are made to normalize the result. In multiplication, shifts are made to left justify the product to preserve as much accuracy as possible.

A complete pass through the program takes about 20 milliseconds using fixed-point computations and about 60 milliseconds using simple high speed floating-point calculations.

ELECTRONIC MOTOR CONTROLLER

Since the permanent magnet motors of the Rancho electric arm do not have good speed control using a variable supply voltage, and proportional control of motors using linear amplifiers results in high power dissipation in the drive circuitry, a variable frequency pulse drive is being used. The computer is programmed to produce a series of pulses with the duty cycle required to obtain the desired speed. The pulse train controls a solid state electronic switch which connects the motor to the battery with the appropriate polarity

TABLE I—Signed Floating-Point Operation Times

Operation	Time Required When No Shifts Needed (Microseconds)	Additional Time For Each Shift Required (Microseconds)
ADD	90	13
SUBTRACT	110	13
MULTIPLY	190	11

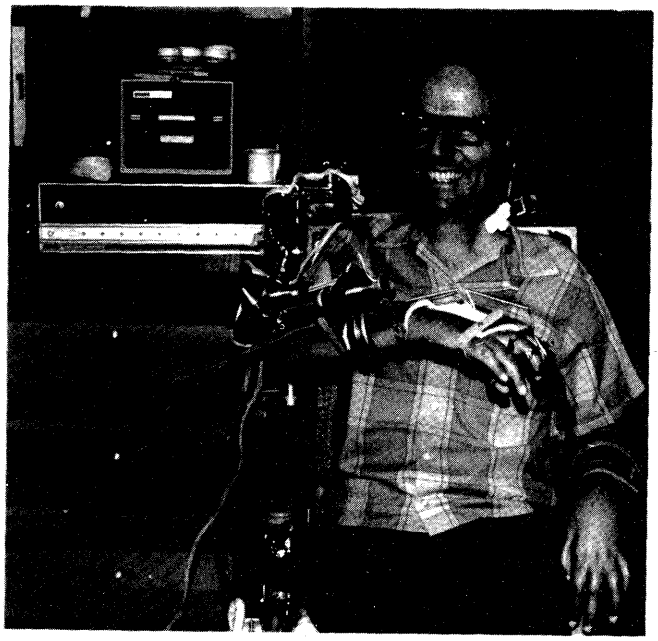


Figure 7—Patient using ocular control system

during each pulse. Since the pulse sequence for each joint is determined by the computer, it is easy to experiment with different strategies of pulse-frequency or pulse-width modulation.

PATIENT TESTING

The prototype system being used for preliminary patient tests is shown in Figure 7. The patient is shown operating the Rancho electric arm by use of the ocular transducer. The minicomputer used to perform the coordinate transformations and control the arm aid motors is shown in the background.

ACKNOWLEDGMENTS

This research was supported in part by Grant AM-10763 from the National Institute of Health, Department of Health, Education and Welfare, Washington, D. C., and carried out in cooperation with Rancho Los Amigos Hospital, Downey, California and Craig Rehabilitation Hospital, Denver, Colorado.

REFERENCES

1 J R ALLEN
The Rancho electric arm
Record of the Third Annual Rocky Mountain Bioengineering Symposium pp 79-82 May 1966

2 J T SCHWARTZ M L MOE C A HEDBERG
A coordinated motion controller for an electric arm orthesis
Record of the Fifth Annual Rocky Mountain Bioengineering Symposium pp 44-48 May 1968

3 J T SCHWARTZ M L MOE
A coordinated motion controller for the Rancho electric arm
Record of the Sixth Annual Rocky Mountain Bioengineering Symposium pp 85-86 May 1969

4 M L MOE J T SCHWARTZ
A coordinated, proportional motion controller for an upper-extremity orthotic device
Proc of the Third International Symposium on External Control of Human Extremities pp 295-305 Dubrovnik Yugoslavia August 1969

5 J T SCHWARTZ M L MOE
Ocular safety considerations for divergent infrared sources in the near field
Record of the Eighth Annual Rocky Mountain Bioengineering Symposium pp 157-162 May 1971

6 T J ZEBO
Myoelectric control of the Rancho electric arm
Proc of the 21st Annual Conference on Engineering in Medicine and Biology November 1968

7 D E WHITNEY
Resolved motion rate control of manipulators and human prostheses
IEEE Transactions on Man-Machine Systems Vol MMS-10 No 2 pp 47-54 June 1969

8 D E WHITNEY
Coordinated control of prosthetic arms
Proc of the 23rd Annual Conference on Engineering in Medicine and Biology p 237 1970

9 P D LAWRENCE W C LIN
Statistical decision making in the real-time control of an arm aid for the disabled
IEEE Transactions on Systems Man and Cybernetics Vol SMC-2 No 1 pp 35-42 January 1972

10 M M GAVRILOVIC M R MARIC
An approach to the organization of the artifical arm control
Proc of the Third International Symposium on External Control of Human Extremities pp 307-322 Dubrovnik Yugoslavia August 1969

11 B SINGH
Design of an electronic controller for an upper extremity orthosis
Denver Research Institute Research Report DRI #2445 March 1968

12 F J GREEB
Equations of motion for control of an upper extremity splint structure
MS Thesis Department of Electrical Engineering University of Denver May 1970

Computer aiding and motion trajectory control in remote manipulators

by A. FREEDY and J. LYMAN

University of California
Los Angeles, California

INTRODUCTION

This paper presents an advanced version of a learning system for operator aiding in artificial limbs and remote manipulator control. The detailed mathematical developments of the technique and results of preliminary work are contained in previous publications.

The nature of this paper is expository and the goal is to introduce the approach and provide a general description of the concept.

The control concept incorporates the sharing of control responsibility between the operator and a separate automaton able to observe the patient responses, learn part or all of the task at hand, and take appropriate control actions. Its purpose is to relieve the operator of routine or exacting control requirements and reduce his information handling load.

A computer is incorporated into the control loop. The computer program contains a learning program which is capable of controlling the manipulator autonomously. Initially all control actions are generated by the operator. As learning occurs the computer begins to participate in the control process and take over control responsibility. The computer based system has been termed as Autonomous Control Subsystem (ACS).

Figure 1 illustrates how such a computer system functions in the man/machine control loop. The major components of this relationship are defined as follows:

(a) *Computer Aiding* is provided by a computer placed parallel to the human operator in the man/machine control loop. The computer aids by making and displaying control decisions, and by supplying autonomous control inputs to the machine system.

(b) *Adaptive Decision Making and Control* comes from a trainable "machine learning" algorithm programmed on the computer. Various sensors allow the program to observe operator performance and its results, and to optimize control decisions accordingly. The computer learns complicated control strategies, can be pretrained for future tasks, and forgets unused actions.

(c) *Decision Information* is presented to the operator continuously by the computer. The amount and type can very with the application, and may include such factors as the degree of confidence in a computer decision, the planned action and the probable outcome, etc.

(d) *Allocation of Function* between the operator and the computer is made on the basis of the particular system and task, the immediate processing load, the decision information, etc. The operator retains the capability to override computer decision, and in fact, operator overrides help train the adaptive component. In future systems the operator may also adjust the adaptive parameters of the computer program.

Systems of this type constitute bona fide examples of the man/computer "symbiosis" heralded a decade ago by Licklider,[2] as Parsons[3] points out, long remained little more than a catchword.

Other related work is being pursued by numerous groups. This work is directed primarily toward achieving independent machine perception, mobility, and manipulation.[4,5,6] Examples are development of scene analysis techniques, "hand-eye" control programs, and robot mobility programs. Related work involves machine problem solving per se and interactive computer languages.[7,8]

ACS System Operation

Manipulator control with the ACS involves two main control loops, an external loop, which incorporates the operator and his means of feedback, and an internal

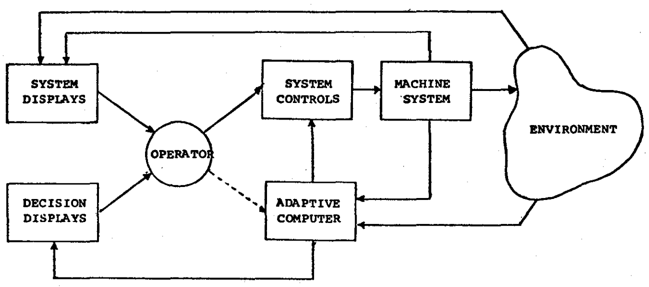


Figure 1—Adaptive computer aiding in man/machine systems

loop, which contains only the autonomous control subsystem (as shown in Figure 2). The manipulator responds to either the operator or the autonomous subsystem.

Initially, the autonomous control subsystem (ACS) acts as a passive observer, trying to "understand" how the operator controls the manipulator and developing an "awareness" of the operating environment. In this phase, the ACS defines the relationship between the operator's responses and the external world.

After the acquisition of this passive experience, the internal loop begins to participate in the control of the manipulator. In this second phase, the operator acts mainly as an instructor, letting the internal loop (loop B) control the manipulator whenever possible, and correcting its decisions when they are wrong. In time, for certain classes of tasks, the function of the operator may be reduced to that of an initiator and inhibitor, who provides occasional start and override commands.

In such an arrangement, the decision load associated with controlling the manipulator is substantially reduced, giving significant advantages in completion time, efficiency of manipulator use, and operator satisfaction.

The ACS is designed with the ability to forget what it has learned, and learn new tasks in place of old unused ones. This feature provides the adaptive capability to change behavior in response to changes in the operator's control policies and in the environment. It makes the ACS a powerful tool, applicable to a large variety of man/machine systems.

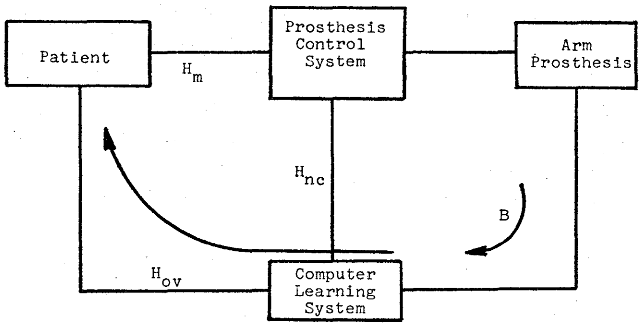


Figure 2—System diagram

Theoretical basis

The theoretical basis for the ACS is the maximum likelihood decision principle.[9] Its structural organization is a conditional probability matrix relating future states of the manipulator devise to its past and present states.[10]

Spatial movement of a manipulator is non-random for practical tasks. That is, patterns of movements in the past lead to predictable movements in the future. In the ACS prediction is based on the likelihood of occurrence of a particular position in space, or of a movement path, computed from the conditional probability matrix.

Maximum likelihood was chosen for the ACS over various other possible classification systems because it has several significant advantages. These include:

- Training is rapid and relatively simple
- Decision strategy can be changed while the system is active
- Classification categories are not restricted to disjoint sets

As the ACS acquires more and more information about previously observed states, the current state, and the next observed state, of the teleoperator it controls, the conditional probability matrix stabilizes, and the ACS achieves its control decisions with a higher level of confidence. Figure 3 illustrates the system organization, and shows how ACS experience leads to "reward" or "punishment" of the probability (P) matrix.

In one sense, the ACS is a redundancy machine, able to extract redundant aspects of a task even if they are hidden in a scatter of apparently random motions. In manipulatory operations the element of redundancy is quite high. Movements from one location to another tend to be repeated, specific paths are often followed, and certain movement spaces are largely avoided. This makes the ACS a shrewd predictor of teleoperator action.

Redundancy is highest in repetitive operations, and the ACS learns extremely fast under these conditions. But it is important to understand that the ACS organization allows it to comprehend much more subtle task factors after long-term operation. For example, as the ACS builds up a depth of experience with a certain general class of tasks, learning of specific subtasks occurs much more rapidly than it did initially. Furthermore, the ACS is able to move among the separate subtasks without losing its adaptation to each.

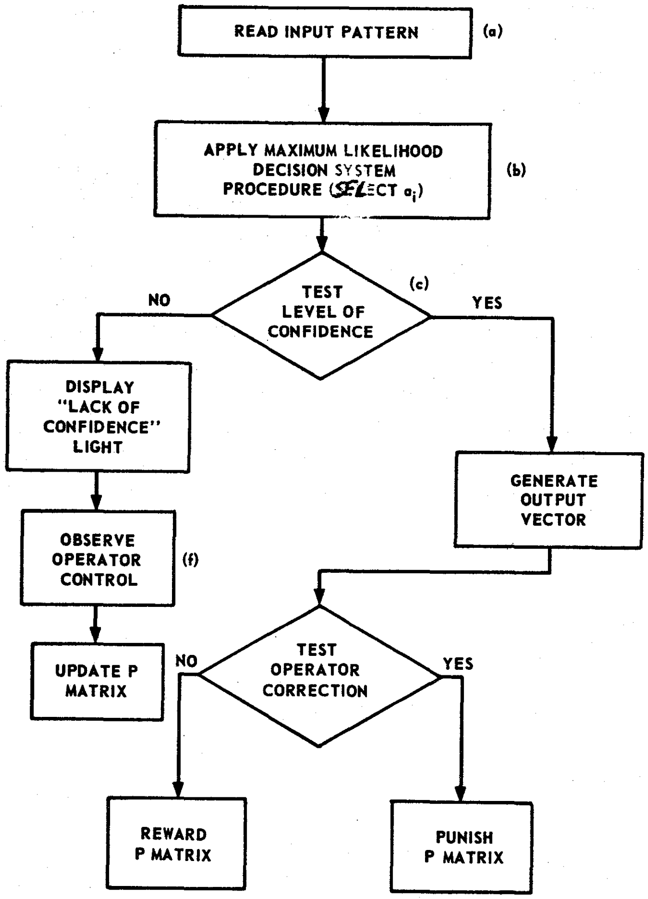


Figure 3—System organization

Likewise, the ACS is able to recognize patterns of movement, of implicit movement "rules," and adapt to changes in such patterns. An operator placing objects sequentially in locations A, B, C, D, . . . will find that the ACS learns to predict . . . E, F, and so on, even though it has not yet observed that particular movement.

Through a built-in function termed "List Control," the ACS is also able to recognize repetitive sequences (or lists) of teleoperator movements, add these lists to the probability Matrix decision space, and retain them as long as they are needed. Thus, the ACS can at times handle a complete subtask as a single decision. There are marked advantages to this approach in practical machine control.

The power of this approach is in the ability of the ACS to operate with limited amount to input data. By using long-term experience it is able to operate autonomously with a minimum amount of sensory cues. It is difficult to gauge the limit of ACS capability when implemented on more powerful, specialized computing systems, and after extensive training with practical machine tasks.

System implementation

A first operational system of the ACS with a remote manipulator has been reported in an earlier publication. Experiments indicated the feasibility of the technique; with a relatively short training period, the ACS was able to initiate the majority of the control decisions necessary to perform a series of manipulative tasks.

The system presented here is an advanced version of the ACS with added capabilities of complete trajectory motion control. The computer specifies a unique set of points in space through which the hand must pass in translation between two points in space. This is in contrast to our earlier system that included only point to point control commands. This capability is advantageous for manipulative tasks, where the environment of operation contains obstacles.

The technique is amenable to prosthesis and manipulator control and has the following properties:

1. Trajectories can be learned from a human operator while he controls the arm in actual tasks. Psychologically the operator can feel he is personally in control whenever the system takes over.
2. Trajectories can be changed as the environment of operation and tasks are changed.
3. The number of trajectories that the system can learn to generate is not constrained by the available computer memory.
4. The learning system can be implemented with currently available mini- or micro-computers, thereby greatly increasing the practicality of the system.

The approach utilizes the learning properties of the computer system to learn typical trajectories generated by the operator. Under this configuration the trajectory of movement of the arm in moving between two points is broken into a set of movement segments. These segments consist of a set of elementary directions which translate the hand between neighboring points. In directing the hand to an end point a sequence of decisions will be performed to determine the instantaneous direction of the movement trajectory. For each submovement segment that is selected (between two consecutive points) a decision will be required. This is accomplished in the following way:

The decision space is illustrated in Figure 4. The three dimensional space of motion is shown at each arbitrary

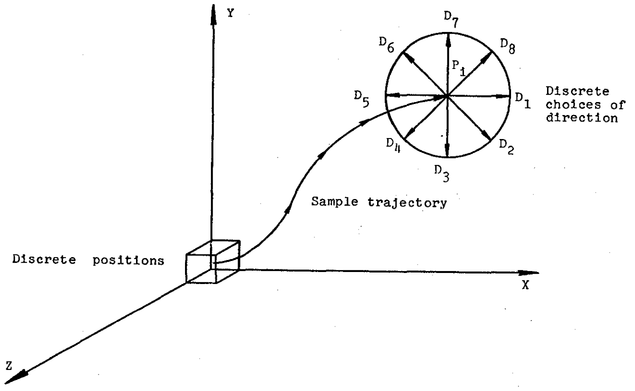


Figure 4—Directional decision space

point in space, Pi. The hand can move to any of 26 directions. These directions include the basic six degrees of freedom along each of the Eucledean axis (x, y, & z) and along a direction which falls symmetrically between each of the axes.

By setting the elementary decision to be a specific direction of motion rather than an absolute location the memory requirements are significantly reduced while the capabilities of the system increases. Since the decision space is constrained to 26 choices the elementary decision computation time is fixed and independent of the size of the allowed space of motion.

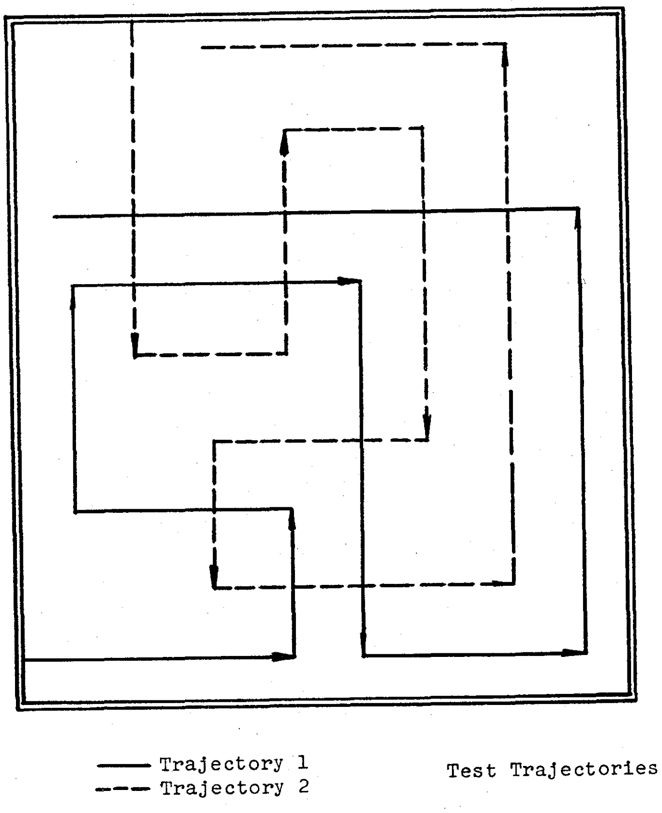


Figure 5—Test trajectories of motion

Refinement of motion accuracy increases the requirement of computer memory in additive fashion rather then multiplicative. This provides the capability for increasing motion accuracy at minimum memory cost.

A prototype trajectory learning system has been developed and implemented on an Interdata Model 70 computer. The system was tested in a simulated 2-dimension environment using a 25 × 25 cm oscilloscope screen. A 2 degrees of freedom, second order system was used to simulate motor motions in two degrees of freedom. The operator trained the learning system by a joy stick controller, moving the curson along a trajectory of motion while the computer observed and learned. As the probability of correct response reached the level of confidence, the computer took over control automatically.

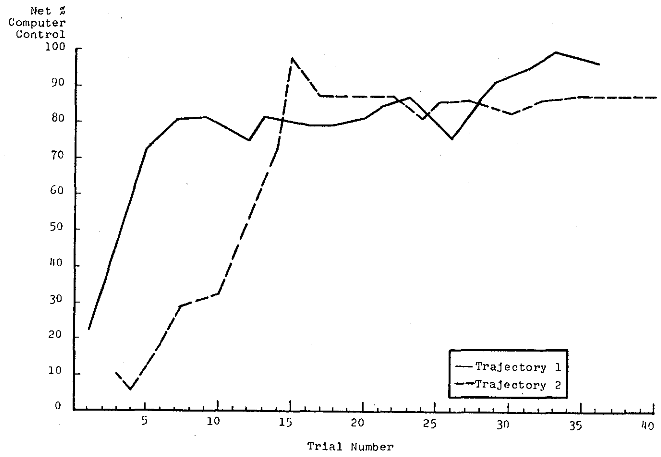


Figure 6—System learning for two trajectories intersecting at single points

The memory requirements of the prototype program included 4K of 16 bit words of memory. 2K were used for storage of the program and 2K were used to store the conditional probability parameters.

Figure 5 illustrates complex trajectories of motion which the system was trained to perform. As can be seen the trajectories intersect each other at 10 points. The trajectories were performed in a random order. Figure 6 describes the performance of the learning system in terms of percent of correct control decisions at each complete trial of the trajectory. The figure illustrates system performance over 40 trials of operation. The ordinate describes the percent of correct decisions while the abscissa represents the trials of operation. As shown, after 10 trials the computer took over more than 80 percent of the control. Inspection of the first trajectory reveals that performance monotonically increases to a level of about 80 percent and then fluctuates around this value. These fluctuations are due to the intersections between the trajectories. Each time the trajectories intersect each other relearning occurs, since the decision regarding a direction of travel is partially based on the location. As two trajectories intersect they have a common decision data point with different decision outcomes.

Human factors aspects

The concept of sharing control responsibility between the operator and a learning system introduces a new dimension to the man-machine relationship. This aspect of the concept must be further explored before optimum allocation of control function can be made.

Factors of importance to this new relationship include the operator approaches his shared task, as well as how closely the ACS is adjusted to fit the operator's capabilities. For example, the learning rate of the machine is adjustable—the basic question is how closely this rate should match the normal operator's learning rate, and how variations in machine learning rate affects the operator's performance.

A related problem is the Feedback aspect of the system. Here the question is what type of feedback the operator must have from the learning system (in addition to the normal required feedback which is associated with operator control). There are a number of possibilities, among them is to provide a display which indicates to the operator the level of experience the learning system has acquired. Such information can be transformed into a certainty measure which will indicate to the operator what decision is to be made, and its level of certainty.

In addition, practical control situations in remote manipulators will most likely involve remote viewing, and may also involve time-delays, such as those encountered in extraterrestrial transmission. The influence of these factors will have to be examined. The objective will be to develop controls, displays, and on operating strategy which allows the operator to take maximum advantage of the help promised by adaptive aiding in various applications.

REFERENCES

1 A FREEDY F HULL L F LUCACCINI J LYMAN
A computer based learning system for remote manipulator control

2 J C LICKLIDER
Man-computer symbiosis
IRE Transactions on Human Factors in Electronics 1960 1 pp 4-10

3 H M PARSONS
The scope of human factors in computer-based data processing systems
Human Factors 1970 12 2 April 1970 pp 165-175

4 M L MINSKY
An autonomous manipulator system
Project MAC Progress Report III MIT July 1970-July 1971

5 R PAUL
Trajectory control of a computer arm
London Artificial Intelligence Conference Imperial College London September 1971

6 N J NILSSON B RAPHAEL
Preliminary design of an intelligent robot
Computer and Information Science II Academic Press Inc New York 1967

7 D J BARBER
MANTRAN: A symbolic language for supervisory control of remote manipulators
S M Thesis MIT 1967

8 D E WHITNEY
State space models of remote manipulation tasks
Massachusetts Institute of Technology PhD Thesis January 1968

9 N J NILSSON
Learning machines
McGraw-Hill Book Company New York 1965

10 A M UTTLEY
Conditional probability machines and conditional reflexes
Annals of Math Studies 34 pp 253-273

A robot conditioned reflex system modeled after the cerebellum

by JAMES S. ALBUS

National Aeronautics and Space Administration
Greenbelt, Maryland

INTRODUCTION

Most modern theories of behavior involve a hierarchical structure whereby low level behavioral units are controlled or manipulated by higher centers so as to produce characteristic patterns of movement. In the simplest life forms, low level behavioral units may consist of simple reflex arcs with very little higher level control. In intermediate forms, the low level behavioral units may be relatively complex in themselves and subject to sophisticated control from higher centers. In the most advanced nervous systems, higher centers may themselves be arrayed in a hierarchical structure, with each level monitoring activity and exerting control over the levels beneath it.

It has been suggested[1,2,3] that the brain has a repertoire of behavioral units arranged much as the keys of a piano. Friedman suggests that a higher level "selecting mechanism activates these behavioral units to produce complex behavior just as an accomplished piano player produces a Beethoven sonata from his simple keyboard."

Theoretical workers in the behavioral sciences have suggested a number of ways in which these selecting mechanisms might interact to choose the proper behavioral units for the task to be accomplished. Just what these behavioral units are, however, or how they are specifically controlled, has been an open question.

There have been, of course, a number of hypotheses concerning the structure and function of reflex arcs. At a very simple level, such as the motor system "gamma loop," these models have been convincing. However, at more complex levels, theories such as Hebb's cell assemblies[4] have been completely unsuccessful in providing substantive explanations for behavioral phenomenon. Sufficient quantitative data concerning the anatomy and physiology of complex brain structures has, until recently, simply not been available for formulating precise models with convincing properties. In the absence of data, most models have been vacuous conjections.

In the past 8 to 10 years, however, the electron microscope and refined techniques of microneurophysiology have revealed quantitative data of considerable detail concerning the structural and functional organization of the brain, particularly in the cerebellum. A great deal of the physiological data about the cerebellum has come from an elegant series of experiments by Eccles and his co-workers. These data have been compiled along with the pertinent anatomical data, in book form by Eccles et al.[5] This book set forth one of the first reasonably detailed theories on the function of the cerebellum.

Shortly after the publication of Eccles' book, another theory was developed by two different researchers working independently. Marr[6] published his Theory of Cerebellar Cortex in 1969 and shortly thereafter the present author[7] published a Theory of Cerebellar Function. Recently this theory has been developed further and reduced to computer software for the control of a mechanical manipulator.

FUNCTION OF THE CEREBELLUM

Although the Theory of Cerebellar Function was developed largely from neurophysiology and anatomical evidence, its reduction to computer software can be explained without detailed knowledge of the biological literature.

The cerebellum, along with the higher level brain centers which control it, can be thought of as a type of finite state mzchine.

$$M = (S, I, O, \delta, \lambda)$$

where

S is a finite non-empty set of states

I is a finite non-empty set of inputs
O is a finite non-empty set of outputs
$\delta:SXI\rightarrow S$ is the transition function
$\lambda:SXI\rightarrow O$ is the output function

The "set" (or state) of the higher level brain centers determines the state S of the cerebellum. The sensory signals from various nerve endings in the limbs being controlled provide input I. The combination of I impinging on the cerebellum in state S, produces output O. The output function $\lambda:SXI\rightarrow O$ corresponds to a reflex arc. In the cerebellum the function λ is defined, and may be altered, through the process of learning.

The transition function $\delta:SXI\rightarrow S$, although undoubtedly of great importance to theories of higher level perception and intelligence, is considered beyond the scope of the elemental reflex level control functions being addressed in this paper. We are here considering merely how the cerebellum can be put into state S by the higher level centers, and then act as a reflex arc which transforms input I into output O under the operation λ. We will also discuss how λ can be altered through training.

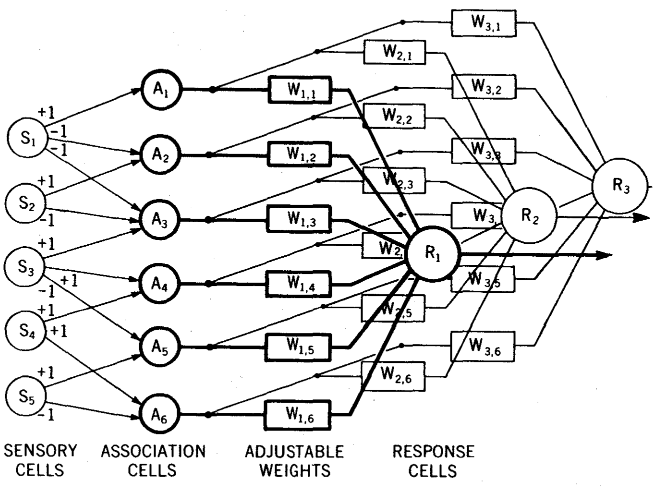


Figure 1—Classical Perceptron. Each sensory cell receives stimulus either +1 or 0. This excitation is passed on to the association cells with either a +1 or −1 multiplying factor. If the input to an association cell exceeds 0, the cell fires and outputs a 1; if not, it outputs 0. This association cell layer output is passed on to response cells through weights $W_{i,j}$ which can take any value, positive or negative. Each response cell sums its total input and if it exceeds a threshold, the response cell R_j fires, outputting a1; if not, it outputs 0. Sensory input patterns are in class 1 for response cell R_j if they cause the response cell to fire, in class 0 if they do not. By suitable adjustment of the weights $W_{i,j}$ various classifications can be made on a set of input patterns.

(Figures 1, 2, and 3 reprinted by permission from Mathematical Biosciences 10, 1971)

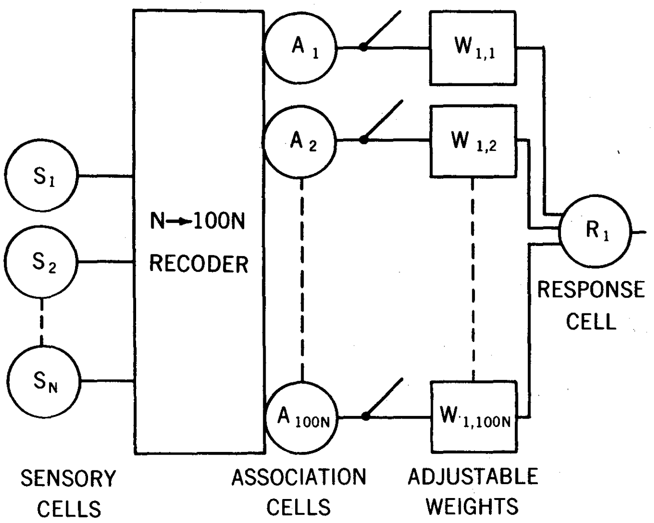


Figure 2—N, 100N Expansion Recoder Perceptron. The association cell firing is restricted such that only 1-2 percent of the association cells are allowed to fire for any input pattern. This Perceptron has a large capacity and fast learning rate, yet it maintains the number of association cells within limits reasonable for the nervous system

Input I enters the cerebellum via mossy fibers from the periphery. (The engineer unfamiliar with anatomical nomenclature must excuse the quaint terminology. Many terms, like mossy fiber, were coined by early anatomists over a century ago. Peering through their crude microscopes and seeing fibers resembling moss, they merely called them as they saw them. In other instances, features such as Purkinje cells were named after the first investigator who observed them.) All mossy fiber input enters a section of the cerebellum called the granular layer. In the granular layer, information carried by the mossy fibers in the form of pulse interval (or frequency) modulation is transformed into information carried on parallel fibers. The important feature of this transformation is that there are from 100 to 1000 times as many parallel fibers coming out of the granular layer as there are mossy fibers going in. This implies that in the granular layer, information is recoded. The evidence seems to indicate that the granular layer transforms mossy fiber information in the frequency domain into parallel fiber information in the spatial domain. The theory predicts that only a very few (i.e., about 1-2 percent) parallel fibers are active for any given pattern of pulse frequency modulation on mossy fibers.

Output from the cerebellum itself is via Purkinje cells. The theory predicts that Purkinje cells perform a weighted summation of parallel fiber activity analogous to the way in which a Perceptron response cell performs a weighted summation on association cell firings. Thus

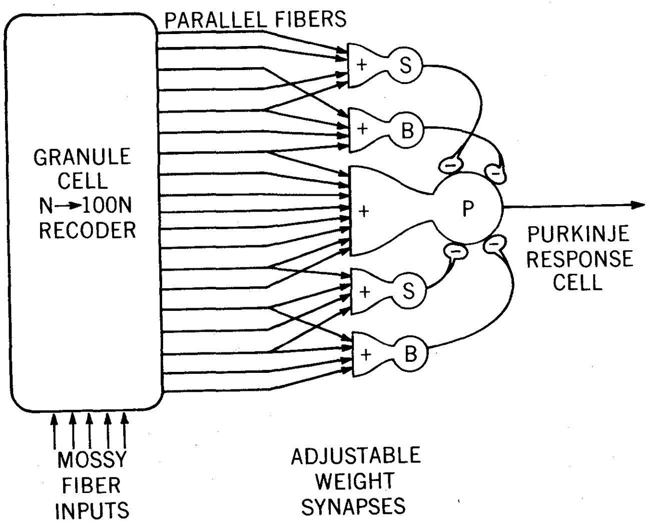


Figure 3—Cerebellar Perceptron: *P*, Purkinje cell; *B*, basket cells; *S*, stellate *b* cells. Each Purkinje cell has inputs of the type shown

the cerebellum can be considered to be a form of Perceptron where mossy fiber input is analogous to sensory cell firings. The granular layer corresponds to the interconnection network between sensory cells and association cells, parallel fibers correspond to association cell outputs and the synaptic connections between parallel fibers and Purkinje cells correspond to the variable weights. The Purkinje cells themselves correspond to the Perceptron response cells. This analogy can be seen in Figures 1, 2, and 3. Figure 1 is the classical Perceptron. Figure 2 shows the classical Perceptron modified to conform to the anatomical fact that the cerebellar granular layer contains more than 100 times as many parallel fibers as input mossy fibers. Figure 3 extends the Perceptron analogy to take into consideration the fact that in the nervous system certain types of cells are excitatory and other types are inhibitory, but no type is both. Thus, in order for the cerebellar Perceptron to have both positive and negative valued weights connecting parallel fibers to Purkinje cells, some intermediary cell types (i.e., *B*, basket cells, and *S*, stellate b cells) are necessary. To an engineer, these intermediary cells are inverters.

There is one important difference between the classical Perceptron and its cerebellar counterpart. The classical Perceptron typically accepts only binary input signals, performs an analog weighted summation, compares this sum with a threshold, and responds with a binary output. The cerebellar Perceptron, on the other hand, accepts input which, although consisting of binary pulses, contains information in the form of pulse frequency modulation which is essentially analog in nature. This analog data is recoded from the frequency domain to the spatial domain by the granular layer. The Purkinje response cell, at least to a first approximation, can be considered a linear summation device. It performs no thresholding in the sense of a classical Perceptron response cell. Purkinje cells are typically spontaneously active at some steady-state output rate. A weighted summation of parallel fiber activity merely increases or decreases the frequency of the Purkinje output pulse train. Thus, both outputs and inputs to the cerebellum should be considered to be analog signals coded into pulse frequency modulation.

Learning

The cerebellum is hypothesized to learn by an error correction system similar to Perceptron training algorithms. Each Purkinje cell is contacted by a single climbing fiber. These climbing fibers are hypothesized to carry the information necessary to adjust synaptic weights in an error correcting manner. Climbing fibers carry information from higher motor centers as well as centers of emotional reward and punishment.[2] These higher centers presumably are able to sense conscious motor commands, compare these conscious commands against the cerebellar reflex motor output, and correct the cerebellar output when it deviates from what the higher centers consider to be satisfactory performance. This correction takes place by adjusting the synaptic weights between active parallel fibers and erroneously responding Purkinje cells. The weights are adjusted so as to null the difference between what conscious centers send to the motor system, and what the cerebellar reflex arc produces. Thus, as training proceeds, more and more of the routine motor control can be relegated to the cerebellar reflex arc, and higher centers are then free to concentrate on other matters.

This corresponds to the common experience which everyone has had when learning a new motor skill. At first, a task such as driving an auto, playing a musical instrument, or roller skating requires a great deal of conscious concentration. However, as learning proceeds, more and more of the new motor skill comes under reflex control, and less conscious mental effort is required. This presumably is the process of training the cerebellum (and other similar subconscious motor centers) to take over the repetitive and routine tasks which can be controlled by reflex responses.

The cerebellum thus can also be viewed as a memory. The mossy fibers constitute the address lines. The climbing fibers constitute the data storage inputs. And the Purkinje cell outputs correspond to the contents of the memory. This is illustrated in Figure 4. The mossy

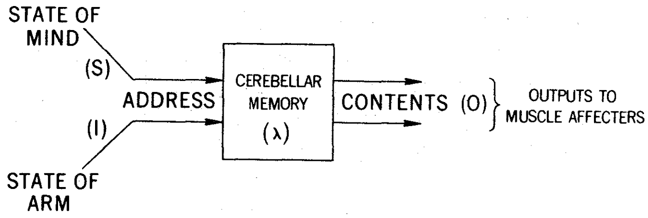


Figure 4—Cerebellar Memory. The state S of higher brain centers is communicated along with the input I from peripheral proprioceptors to the cerebellum via mossy fibers. S and I together constitute an address. Data to be stored arrive via climbing fibers

fiber input constitutes an address. The Purkinje cell response corresponds to the memory contents. Each Purkinje cell output can thus be considered a separate memory bank. By this means the cerebellum achieves the redundancy which is so characteristic of circuitry in the brain.

It, of course, is obvious that if mossy fiber address lines are essentially analog in nature, there exists an enormous number of possible addresses. If we assume that there are N mossy fiber address lines, and each mossy fiber can carry an analog signal with 50 distinguishable values of pulse frequency, then we have 50^N possible addresses. If we consider that each square millimeter of granular layer has approximately 5×10^4 mossy fibers entering it, we clearly have a potentially enormous number of addresses. However, one must remember that if the world is subjected to a state-space analysis, there exists an equally enormous number of possible states-of-the-world. Mossy fiber input from sensory receptors in the limbs are essentially reporting the state of the limbs. Since people and animals are able to cope with the infinity of possible states in the real world, it is clear that somehow these states are grouped into a manageable number of sets of states. States within such groupings are for all practical purposes equivalent. So too, the virtual infinity of possible mossy fiber addresses are grouped into sets of essentially equivalent addresses. This grouping is accomplished by the granular layer. The granular layer performs a transformation such that if two mossy fiber addresses are within an equivalence group, the same pattern of parallel fiber outputs will occur.

The mossy fiber input can be considered a vector

$$I = (mf_1, mf_2, mf_3, \ldots mf_N)$$

where mf_i is the firing rate of the i-th mossy fiber.

We can define similarity between mossy fiber patterns in terms of the Hamming distance H_I between input vectors I and I'.

$$H_I = \sum_{i=1}^{N} | mf_i - mf_i' |$$

The mossy fiber input vector I is transformed by the granular layer into a parallel fiber vector

$$J = (pf_1, pf_2, pf_3, \ldots pf_{100N}.)$$

where pf_i is the firing rate of the i-th parallel fiber.

The theory hypothesizes that at any instant of time only about two percent of the pf_i firing rates are non-zero. Thus, the vector J is a very sparce vector. The principal feature of this transformation is the conversion of mossy fiber patterns in the frequency domain to parallel fiber patterns in the spatial domain. Parallel fibers thus are hypothesized to code information in terms of the specific set of parallel fibers which have non-zero firing rates. We can define a set

$$L = \{pf_i \mid pf_i \text{ has a non-zero firing rate}\}$$

We can then represent similarity between two parallel fiber patterns J and J' in terms of the intersection

$$L \cap L'$$

The granular layer is hypothesized to perform such that if H_I is small, $| L \cap L' |$ will be large, and as H_I grows large $| L \cap L' |$ will approach zero. This implies that training for dissimilar tasks (i.e., such that H_I is large) will produce very little interference. Weights adjusted for pattern I will be entirely different from those adjusted for pattern I' because $| L \cap L' |$ is zero. However, for similar patterns, training will generalize. Similar mossy fiber patterns (i.e., H_I small) will cause many or most of the same weights to be adjusted because $| L \cap L' |$ is large. Thus, the cerebellum need not be trained to cope with every possible mossy fiber address corresponding to every possible state of the arm. Instead, training over a small but representative sample of the possible states will suffice.

An electro-mechanical model

Consider now how such a model of the cerebellum can be reduced to computer software. An IBM 1800 computer was connected to a Rancho Los Amigos arm with seven degrees of freedom. Each degree of freedom was driven by a separate motor. Each motor amplifier was controlled by a computer model of a single Purkinje summation cell. Each Purkinje summation in the computer thus represents a large number of Purkinje cells in the real cerebellum; some activating flexar muscles, others activating extensor muscles.

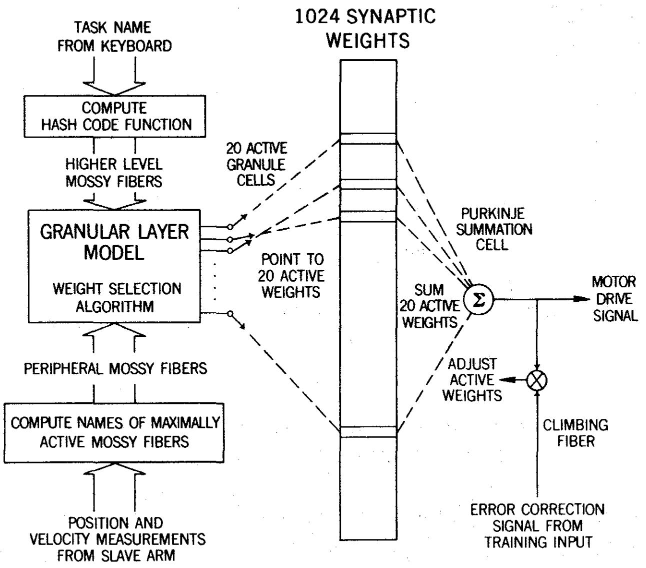


Figure 5—Computer model of cerebellar Perceptron

Each of the seven Purkinje summation cells in the computer are related to a table of 1024 synaptic weights, as shown in Figure 5. These weights are adjustable over the range −32767 to +32767. According to the theory, only two percent of the parallel fibers, and hence two percent of the synaptic weights, are activated at any one time. In the computer model, the granular layer selects 20 out of the 1024 weights to be active. The Purkinje cell then sums the values of these 20 active weights. This summation is the Purkinje cell output.

In the model, mossy fiber inputs convey information concerning the position and velocity of each joint. This information constitutes an address which is converted by the granular layer into a set of 20 active parallel fibers. These 20 parallel fibers connect to 20 active weights which are summed by the Purkinje cell. The value of this summation can be considered to be the "contents" of the memory location addressed by the mossy fiber pattern.

For any position-velocity state of the arm, each Purkinje summation cell delivers a drive voltage to the actuator motors. If this voltage is not appropriate to the task being attempted, it can be modified by adjusting the 20 active weights in each Purkinje summation. This is the training mode. When the arm is being trained to perform a particular task, the weights selected by the granular layer are adjusted by the training algorithm to follow the instructions being generated by the teacher. The teacher in the model is a master arm worn by a human operator. The training operation begins by the operator entering the name of the task to be learned on a keyboard. This name corresponds to the psychological "set" or state of the higher centers in the brain. This determines the state S of the cerebellum. For example, the number 0101 on the keyboard might correspond to the task "reach-out." The operator would then proceed to teach the cerebellar model by performing a reach-out motion with the master arm. At closely spaced intervals along the reach-out trajectory the controlled arm position is compared against the master arm position. Whenever a discrepancy is detected, the weights connected to the 20 active parallel fibers are adjusted so as to drive the motors in a direction which will null the difference.

By this means, the memory stores the proper motor drive voltage for each position-velocity state along the desired trajectory. Repeated training can store the proper corrective voltage outputs for other states to either side of the desired trajectory. The generalization properties of the memory make it feasible to train the arm on only a representative sample of the universe of possible states, and still achieve satisfactory performance. The process of training defines the function λ for the universe of input states I encountered in performing the task S.

MODELING THE GRANULAR LAYER

The selection of which set of parallel fibers are active at any instant of time is the function of the granular layer. It is one of the principal hypotheses of the Theory of Cerebellar Function that the manner in which this selection is made gives the cerebellum its unique powers of motor coordination, precision control, and flexibility.

The origin of coordination

It has been experimentally shown[8] that a somatotopic mapping exists from the cerebellar cortex to the muscles of the body. This means, for example, that Purkinje cells affecting the elbow are likely to be physically located in close proximity to each other, and an appreciable distance from those affecting the wrist. Since any single parallel fiber extends only about 1 mm. along a folial ridge, it is quite unlikely that a parallel fiber which contacts an elbow Purkinje will also contact a wrist Purkinje. This implies that the sets of parallel fibers involved in Purkinje summations for controlling specific joints in the model should be disjoint. Thus, in the model, each Purkinje summation will involve a separate set of granule cells and a separate set of weights.

On the other hand, somatopy is much less well defined from the periphery to the cerebellum.[9,10,11,12]

Granule cells which are close neighbors in the cerebellum often have receptive fields at widely separate areas of the same limb or even in different limbs.[13] Mossy fibers enter the cerebellum and ramify diffusely throughout a single folia and even into several different folia. This is not to say that somatopy is non-existent for mossy fiber input, just that it is diffuse and overlapping.

The implication is that Purkinje cells are fairly specific in their control over individual muscles, or synergetic groups of muscles. However, the input to any particular Purkinje cell in the cerebellum, while strongest from its somatopic area in the periphery, is also appreciably strong from other areas of the periphery. Strength of influence from neighboring peripheral areas falls off slowly with distance. Thus, the strongest input to a Purkinje cell controlling the elbow should arrive via mossy fibers from the elbow. However, an appreciable input to the elbow Purkinje should also come from the forearm and shoulder, and to a lesser extent from the wrist and hand. Similar conditions exist for each set of Purkinje cells corresponding to each joint. Input should be strongest from the joint to which a Purkinje projects, and fall off in strength from other joints as a function of distance.

The relevance matrix

In order to model the relative degree of influence which mossy fibers from the various joints have on the sets of granule cells unique to each joint, a relevance matrix is constructed as shown in Figure 6.

The numbers in this matrix indicate relative values. Each row sums to 72. (The number 72 derives from the fact that there are 72 entries in the matrix shown in Figure 8). The first row of the matrix suggests that 30/72 of the mossy films influencing shoulder rotation carry feedback information concerning the state of the shoulder rotation joint, 12/72 carry information concerning the shoulder lift joint, 9/72 concern the elbow rotation joint, 9/72 concern the elbow lift joint, 6/72 concern forearm rotation, and 6/72 concern wrist lift. Similarly for forearm rotation. Row 5 of the matrix indicates that 30/72 of the peripheral mossy fiber input carries information about forearm rotation, 12/72 about wrist lift, 9/72 about elbow lift, 9/72 about elbow rotation, 6/72 about shoulder lift, and 6/72 about finger grasp. Each functional portion of the cerebellum has a different mixture of inputs. In each case 30/72 of the input to the control circuit for each joint is simply feedback information from that joint. The remaining 42/72 of the input carries information concerning related joints.

via Mossy fiber from / Input to Purkinje controlling	Shoulder		Elbow		Forearm	Wrist	Finger
	Rotate	Lift	Rotate	Lift	Rotate	Lift	Grasp
Shoulder Rotate	30	12	9	9	6	6	
Shoulder Lift	12	30	9	9	6	6	
Elbow Rotate	6	6	30	15	9	6	
Elbow Lift	6	6	15	30	9	6	
Forearm Rotate		6	9	9	30	12	6
Wrist Lift		6	6	6	12	30	12
Finger Grasp	6	6	6	6	9	12	30

Figure 6—Relevance Matrix. This matrix represents the relative degree to which input from each joint is relevant to the computation of motor output for each joint

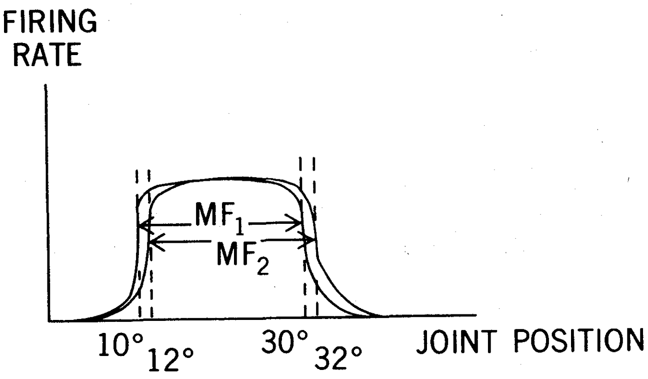


Figure 7—Assumed mossy fiber firing rates plotted against joint position for two different mossy fibers

The mechanism of selection

It seems to be the case[10] that individual mossy fibers fire at their maximal rate when specific conditions exist in specific parts of the periphery. A mossy fiber carrying joint position information will tend to fire at its maximum rate when a specific joint is within a certain range of positions. For example, a typical elbow position fiber might fire at its maximum rate when the elbow joint angle is between 10° and 30°, and at a slower rate otherwise. A different position fiber might fire maximally for elbow positions between 12° and 32°, etc. It has been observed that position fibers fire maximally over some extended range and that considerable overlap exists between the maximal firing ranges of various fibers. See Figure 7.

A mossy fiber carrying information concerning joint velocities will tend to fire at its maximum rate when a particular joint is moving at a rate within a certain range of velocities. Some mossy fibers indicate positive velocities and others negative velocities.

Mossy fibers which fire at their maximum rate are of critical importance if it is true that only 1-2 percent of the parallel fibers are active at once. This 1-2 percent hypothesis implies that for any granule cell to be active a very special set of excitation conditions must be satisfied. Active granule cells must have their input in the upper 1-2 percent of excitation values. Since granule cells have relatively few mossy fiber inputs, each input contributes a large percentage to the total excitation of the cell. It is thus reasonable to assume that for a granule cell to become a member of the very select set of active cells, all, or nearly all, of its mossy fiber inputs must be firing at or near their maximum rate.

From anatomical measurements concerning the numbers and densities of granule cell inputs,[5] and arguments concerning probability of excitation by mossy fiber inputs,[14] it is possible to predict that:

10 percent of active granule cells have 1 input (*s*)
20 percent of active granule cells have 2 input (*s*)
20 percent of active granule cells have 3 input (*s*)
20 percent of active granule cells have 4 input (*s*)
15 percent of active granule cells have 5 input (*s*)
10 percent of active granule cells have 6 input (*s*)
5 percent of active granule cells have 7 input (*s*)

This implies that in a model where 20 granule cells are active,

2 should be a function of 1 mossy fiber (*s*)
4 should be a function of 2 mossy fiber (*s*)
4 should be a function of 3 mossy fiber (*s*)
4 should be a function of 4 mossy fiber (*s*)
3 should be a function of 5 mossy fiber (*s*)
2 should be a function of 6 mossy fiber (*s*)
1 should be a function of 7 mossy fiber (*s*)

Notation for naming

In order to compute the seven sets of 20 active granule cells in a computationally efficient manner, it is convenient to introduce some special notation. First, each mossy fiber entering the cerebellum will be given a unique number. Such a numbering is, in fact, a notation for naming. We will refer to each mossy fiber's number as its name. Thus, mossy fiber #1 is named 1, mossy fiber #2 is named 2, etc. It will also be convenient from time to time to refer to mossy fibers by another convention, or "nickname." 1 will be nicknamed MF_1, 2 will be nicknamed MF_2, etc.

We will now define a classification of mossy fibers called an exclusive set.

> Df: An exclusive set is the set of all mossy fibers such that no two mossy fibers can possibly be maximally active simultaneously.

For example, if

MF_1 is maximally active when the elbow is between 0° and 40°
MF_2 is maximally active when the elbow is between 40° and 80°
MF_3 is maximally active when the elbow is between 80° and 120°
MF_4 is maximally active when the elbow is between 120° and 160°

then $\{MF_1, MF_2, MF_3, MF_4\}$ is an exclusive set.

> Df: A complete exclusive set is an exclusive set in which at least one mossy fiber is always maximally active. For example, the exclusive set given above would be a complete exclusive set if the elbow would never move outside the range 0° to 160°.

It is assumed that each joint has a number of both position and velocity mossy fibers with overlapping ranges of maximal excitation. These can be grouped into complete exclusive sets.

Df: ${}^iP_k{}^j$ is defined as the *k*th complete exclusive set of position-indicating mossy fibers coming from joint *i* and carrying information for Purkinje cell *j*.

Df: ${}^iV_k{}^j$ is the *k*th complete exclusive set of velocity mossy fibers from joint *i* going to Purkinje cell *j*.

Df: ${}^iP_k{}^j$ is the name of the mossy fiber in ${}^iP_k{}^j$ which is maximally active.

Df: ${}^iV_k{}^j$ is the name of the mossy fiber in ${}^iV_k{}^j$ which is maximally active.

For example, if

$${}^1P_3{}^2 = \{MF_{10}, MF_{12}, MF_{15}\}$$

and MF_{15} is maximally active, then

$${}^1P_3{}^2 = 15$$

The *k* subscript indicates different exclusive sets of mossy fibers with overlapping ranges. For example,

${}^1P_1{}^1$ might refer to the set $\{MF_1, MF_2, MF_3, MF_4\}$ such that

MF_1 is maximally active when $0° \leq \alpha < 40°$
MF_2 is maximally active when $40° \leq \alpha < 80°$
MF_3 is maximally active when $80° \leq \alpha < 120°$
MF_4 is maximally active when $120° \leq \alpha < 160°$

and

${}^1P_2{}^1$ might refer to the set $\{MF_5, MF_6, MF_7, MF_8\}$

such that

MF_5 is maximally active when $0° \leq \alpha < 38°$
MF_6 is maximally active when $38° \leq \alpha < 78°$
MF_7 is maximally active when $78° \leq \alpha < 118°$
MF_8 is maximally active when $118° \leq \alpha < 158°$

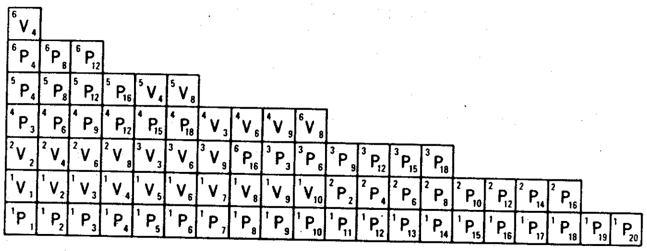
The granular layer matrices

The above notation will now make it possible to compute which 20 granule cells are selected for each Purkinje summation.

As was previously discussed, the statistical distribution of the number of inputs per active granule cell indicates that out of 20 active granule cells, two depend on only one mossy fiber input, four depend on two inputs, etc. This functional relationship can be formulated into a matrix as shown in Figure 8.

This matrix is the granular layer matrix for the shoulder rotation joint. Each space in the matrix is assigned to a complete exclusive set of mossy fibers such that the corresponding matrix element is the name of the maximally active mossy fiber in that set. All of the mossy fiber sets represented in this matrix are carrying information to the shoulder rotation Purkinje summation. In the model there are six other matrices, similar to this one, for the six other Purkinje summations. These matrices contain information concerning the state of the arm as reported by the peripheral mossy fibers.

The numerical values of the elements in these matrices change as the state of the arm changes. A small change in the arm will cause a small number of elements to change in value. A large change in the state of the arm will cause many or all of the elements to change in value.



${}^{6}V_{4}$																			
${}^{6}P_{4}$	${}^{6}P_{8}$	${}^{6}P_{12}$																	
${}^{5}P_{4}$	${}^{5}P_{8}$	${}^{5}P_{12}$	${}^{5}P_{16}$	${}^{5}V_{4}$	${}^{5}V_{8}$														
${}^{4}P_{3}$	${}^{4}P_{6}$	${}^{4}P_{9}$	${}^{4}P_{12}$	${}^{4}P_{15}$	${}^{4}P_{18}$	${}^{4}V_{3}$	${}^{4}V_{6}$	${}^{4}V_{9}$	${}^{6}V_{8}$										
${}^{2}V_{2}$	${}^{2}V_{4}$	${}^{2}V_{6}$	${}^{2}V_{8}$	${}^{3}V_{3}$	${}^{3}V_{6}$	${}^{3}V_{9}$	${}^{6}P_{16}$	${}^{3}P_{3}$	${}^{3}P_{6}$	${}^{3}P_{9}$	${}^{3}P_{12}$	${}^{3}P_{15}$	${}^{3}P_{18}$						
${}^{1}V_{1}$	${}^{1}V_{2}$	${}^{1}V_{3}$	${}^{1}V_{4}$	${}^{1}V_{5}$	${}^{1}V_{6}$	${}^{1}V_{7}$	${}^{1}V_{8}$	${}^{1}V_{9}$	${}^{1}V_{10}$	${}^{2}P_{2}$	${}^{2}P_{4}$	${}^{2}P_{6}$	${}^{2}P_{8}$	${}^{2}P_{10}$	${}^{2}P_{12}$	${}^{2}P_{14}$	${}^{2}P_{16}$		
${}^{1}P_{1}$	${}^{1}P_{2}$	${}^{1}P_{3}$	${}^{1}P_{4}$	${}^{1}P_{5}$	${}^{1}P_{6}$	${}^{1}P_{7}$	${}^{1}P_{8}$	${}^{1}P_{9}$	${}^{1}P_{10}$	${}^{1}P_{11}$	${}^{1}P_{12}$	${}^{1}P_{13}$	${}^{1}P_{14}$	${}^{1}P_{15}$	${}^{1}P_{16}$	${}^{1}P_{17}$	${}^{1}P_{18}$	${}^{1}P_{19}$	${}^{1}P_{20}$

Figure 8—Shoulder Rotation Granular Layer Matrix. This matrix is used to compute which set of 20 granule cells are active for the shoulder rotation joint. Note that input from shoulder rotation position ${}^{1}P_{k}$ and shoulder rotation velocity ${}^{1}V_{k}$ occupy a dominant role in the shoulder rotation matrix. In the model there are six additional granular matrices, one for each of the six other joints

The particular assignments of elements from each joint represented in the matrices are derived, in part, from the relevance matrix in Figure 6. For example, the granular layer matrix for shoulder rotation, shown in Figure 8, has twenty position sets and ten velocity sets making a total of thirty sets of mossy fibers from the shoulder rotation matrix. This corresponds to the fact that the relevance matrix in Figure 6 specifies that 30/72 of the inputs to the shoulder rotation Purkinje should come from shoulder rotation mossy fibers. It is arbitrarily assumed that approximately ⅔ of the inputs from each joint should be position indicators and the remaining ⅓ should indicate velocity.

As can be seen from Figure 8, the assignment of particular sets to particular matrix elements was not done randomly. This was because it was felt that the number of matrix elements was too small to rely on statistical probabilities to give representative importance to the various mossy fiber inputs. Therefore, the various matrices were set up by hand and represent (as does the relevance matrix) the subjective judgment of the author as to which inputs are important to each Purkinje cell for controlling motor outputs. It is important to emphasize, however, that once these matrices are set up they are not changed. This corresponds to a granular layer structure defined by genetically coded interconnections and not structurally altered during an animal's lifetime.

Computation of active granule cell names

These granular layer matrices can now be used to compute which granule cells are active. In each matrix the 20 columns correspond to 20 active granule cells. Columns 19 and 20 correspond to granule cells with only one mossy fiber input. Columns 15, 16, 17, and 18 correspond to granule cells with two inputs, etc. The names of the active granule cells can be computed by the concatenation of elements in the columns of the matrices. For example, in Figure 8, the name of the granule cell computed by column 11 would be ${}^{3}P_{9}{}^{1}{}_{n}$ $\boxed{{}^{2}P_{2}{}^{1}}$ $\boxed{{}^{1}P_{11}{}^{1}}$ If

$$\boxed{{}^{3}P_{9}{}^{1}} = 15, \boxed{{}^{2}P_{2}{}^{1}} = 12, \text{ and } \boxed{{}^{1}P_{11}{}^{1}} = 59$$

then, 151259 is the name of the granule cell computed by column 11. Thus, we have described a method for finding the names of granule cells, given the state of the mossy fiber inputs.

At this point in the discussion, the names we have defined for granule cells are not yet in a particularly useful form nor are they necessarily even unique. The uniqueness problem is rather easily solved by requiring each mossy fiber name to contain the same number of digits. Leading zeros may be employed to accomplish this for mossy fiber names with small numerical values. For example, if it requires three digits to name all the mossy fibers, then mf_1 will be named 001, mf_2 will be named 002, etc. The question of how to utilize the names of granule cells once they are determined is slightly more complicated. Assume, for example, that an active granule cell is named 10956321. How do we find the contribution this cell firing makes to its respective Purkinje output? In the computer it is necessary to locate the weight which connects a granule cell to its respective Purkinje in order to compute its effect. This implies that for each granule cell which is active there must be a pointer which can locate its respective weight in a table of weights. Setting up a table of pointers for each possible granule cell name would be a most tedious job. Fortunately, there is a much simpler technique available. We can instead map the active granule cell names onto the set of integers from 1 to 1024 by means of hash-coding. This may be done quite simply by use of a pseudo-random number generator which uses the numerical value of the active granule cell name as an argument and computes a pseudo-random number in the range 1 to 1024. This pseudo-random number can be considered a new name or "alias" for the active granule cell. This alias can be used directly as a pointer to a table of 1024 weights which connect granule cells to Purkinje cells.

In summary, the following procedure obtains. Each joint has a matrix representing its own peculiar mossy fiber input distribution. In each of the seven matrices we may compute the mossy fibers which are maximally active from measurements of joint position and velocity. The names of these maximally active mossy fibers make up the elements in each matrix. Concatenation of the elements in each column yields the names of active granule cells. Each of these names are used as input to a pseudo-random number generator which maps them onto the integers from 1 to 1024. The result of this procedure is seven sets of 20 integers. These integers point to the 7 sets of 20 weights which are summed by the 7 Purkinje cells. The resulting summations define the output signals which drive the motors for each joint.

Computations for the mechanical arm

In the actual electro-mechanical arm there are, of course, no mossy fibers with overlapping characteristics as in the physiological arm. Instead, each joint has a potentiometer which measures position to a rather high degree of precision. Such a measurement certainly contains all the information which a multiplicity of overlapping mossy fibers would contain. However, the system of overlapping mossy fibers and granule cell-Golgi cell network, produce the phenomenon that if the arm moves slightly, only a few granule cells change from active to inactive, or vice versa; the great majority of granule cells are unaffected. Mossy fibers map their activity into patterns of granule cell activity which are "nearly the same" when the state of the arm is "nearly-the-same." In order for the computer model to capture this "nearly-the-same" property, a method has been devised for converting the potentiometer readings into names of maximally active mossy fibers. These names can then be used as elements in the granular layer matrices.

Rather than attempt to describe the details of these computations in the limited space available here, the interested reader is referred to Reference 14 in the bibliography.

Input from higher centers

Higher level mossy fibers constitute a major source of input to the entire cerebellum. Once these fibers enter the granular layer, they are physically indistinguishable from peripheral mossy fibers. Because of their great numbers, they undoubtedly have a very strong influence on the selection of which granule cells are to be active. This would seem to imply that higher level mossy fibers should be represented in the matrices used to compute the granular layer transfer function. In fact, since the higher level mossy fiber input is so massive, it would seem that it should dominate the granular layer matrices. Surely a change in input on a mossy fiber system which so permeates the entire granular layer should affect, either directly or indirectly, the firing threshold of practically every granule cell in the cerebellum. And so it does. However, there is a simpler way of modeling the influence of higher level mossy fibers than inserting them in the granular layer matrices explicitly. A change in the pseudo-random number generator can model the effects of a very broad and diffuse change in granule cell thresholds throughout the entire cerebellum. Thus, the effect of higher level mossy fiber input can be modeled by assuming the hash-code operation to be under the control of higher centers.

The cerebral cortex is, or course, the place where decisions are made as to what task should be performed by the motor system. The cortex may decide that the arm should perform the task "reach out." This "reach out" task would then be sent to the cerebellum as a specific cerebral mossy fiber firing pattern $\bar{M}_{\text{RO}}$. With the $\bar{M}_{\text{RO}}$ pattern on the cortical mossy fibers and with the peripheral mossy fibers reporting the state of the various joints in the arm, the cerebellar Purkinje cells would tend to produce outputs to drive the motors to "reach out." If these outputs were incorrect, error correction signals via the climbing fibers would cause adjustments in the weights leading from the active granule cells. Thus, the cerebellum would be trained by error correction to correctly perform the task "reach out."

If the cerebral cortex were then to decide that the arm should perform the task "pull back," a new pattern $\bar{M}_{\text{PB}}$ would be sent to the cerebellum via the cortical mossy fibers. This new pattern $\bar{M}_{\text{PB}}$ would change the hash-code function and cause a completely different set of granule cells to be chosen for any pattern of peripheral mossy fiber inputs. Once again the cerebellum could be taught to perform the "pull back" operation by adjusting weights under direction of climbing fiber error correction. Each different task decided upon by the cerebral cortex can be communicated to the cerebellum by a different firing pattern on the cortical mossy fibers. In the model this implies that each different task should be assigned a different hash-coding function.

By this means the cerebral cortex is able to impose high level control on the motor system without worrying about continuous control of each individual muscle. The lower level control functions are carried out by the cerebellum. Only in the case of learning a new motor task, or in case of errors or deviations from the desired performance of previously learned tasks, does the cerebrum need to worry about the detailed control of lower level motor functions.

The cerebellum thus is the repository of detailed motor control sequences which have been previously learned under conscious effort, and which can be called up repeatedly by task name via higher level mossy fiber patterns much as subroutines are called by an executive program. The higher level mossy fibers make it possible for higher centers to control the cerebellar motor system with a macro command language.

REFERENCES

1 K Z LORENZ
The comparative method in studying innate behavior patterns
Physiological Mechanisms in Animal Behavior Symp Soc Exper Biol 4 London Academic Press 1950 pp 221-268

2 J M R DELGADO
Free behavior and brain simulation
International Review of Neurobiology 6 C C Pfeiffer and J R Smythies (Eds) New York Academic Press 1964 pp 340-449

3 L FRIEDMAN
Instinctive behavior and its computer synthesis
Behavioral Science 12 1967 pp 85-108

4 D O HEBB
The organization of behavior: A neuropsychological theory
New York Wiley 1949

5 J C ECCLES M ITO J SZENTAGOTHAI
The cerebellum as a neuronal machine
New York Springer-Verlag 1967

6 D MARR
A theory of cerebellar cortex
J Physiol London 202 1969 pp 437-470

7 J S ALBUS
A theory of cerebellar function
Mathematical Biosciences 10 1971 pp 25-61

8 A BRODAL
Anatomical studies of cerebellar fiber connections with special reference to problems of functional localization
Progress in Brain Research The Cerebellum Vol 25 C A Fox and R S Snider (Eds) Elsevier New York 1967 pp 135-173

9 B HOLMQUIST O OSCARSSON I ROSEN
Functional organization of the cuneo-cerebellar tract in the cat
Acta Physiol Scand 58 1963 pp 216-235

10 J K S JANSEN K NICOLAYSEN T RUDJORD
Discharge pattern of neurons of the dorsal spinocerebellar tract activated by static extension of primary endings of muscle spindles
J Neurophysiol 29 1966 pp 1061-1086

11 A LUNDBERG O OSCARSSON
Functional organization of the dorsal spino-cerebellar tract in the cat
Acta Physiol Scand 38 1956 pp 53-75

12 O OSCARSSON
Functional organization of the spino- and cuneocerebellar tracts
Physiol Rev 45 1956 pp 495-522

13 W T THACH
Somatosensory receptive fields of single units in cat cerebellar cortex
J Neurophysiol 30 1967 pp 675-696

14 J S ALBUS
Control of a manipulator by a computer model of the cerebellum
PhD Thesis University of Maryland 1972

Data base design using IMS/360

by R. M. CURTICE

Corporate-Tech Planning Inc.
Waltham, Massachusetts

TOWARD A DATA BASE DESIGN METHODOLOGY

Data base or file design is the process of specifying how the data is to be located on and retrieved from the various storage media, and what relationships exist among the keys, data elements, records and files of the data base. Occasionally it is useful to distinguish between file design and file engineering. File design is concerned with the logical relationships among file elements, while file engineering deals with physical concerns such as block size, arm movement optimization, and other hardware dependent factors. Although file design necessarily precedes file engineering, very often several iterations between these two are necessary because file engineering considerations suggest a rethinking of many of the file design approaches.

People have been designing data bases with measurable success for as long as there have been random access devices. What eludes us however, is a clear formulation of the precise steps by which the design was constructed. No one has been able to describe a coherent methodology for achieving an optimal or even a good design. Much of data base design thus remains an art.

One prerequisite to the development of a data base design methodology is a clear measure of performance of the resulting design. At first glance we are inclined to measure performance merely in terms of say, daily running time, or total number of accesses, for a given volume of transactions. But often we are quite willing to trade several hours of overnight batch running time to speed up an on-line transaction by a few seconds, or to take an extra disk revolution for a write check to insure data integrity. Moreover, how do we specify a desirable balance between running time and storage size? Or account for periodic reorganizations or future flexibility? It may well be that these factors will have to lie outside the measures associated with initial design methodologies, and a limited objective constructed. Such an objective may be expressed as "given a maximum storage capacity, certain reorganization frequencies, a minimum response time on on-line transactions, etc., then what is the optimum data base design?"

In the end, what we are striving for is something like a deterministic model of data base design, in which the parameters of a particular design situation are input, and a full data base design is output—but we appear to lack the formalisms necessary to describe all the elements involved, as the examination of measures above indicates. Some work has been done on a formal description of a data base structure, and to a degree the COBOL Data Division or something similar would suffice. But very little work has been done on a symbology for transactions, and the resulting "transformations" to the data base they are intended to achieve.

Until such a model, or other data base design formalism, is developed we can only generalize upon past experience in order to construct rules or guidelines toward a design methodology. Several papers present rules of thumb for this purpose.[1–6] These and other discussions of data base design methodology usually make no assumptions about the hardware or software used to implement the design. As more users turn to generalized data base management systems, however, the need arises to identify design guidelines which are specific to a particular system. This paper discusses several data base design guidelines based on the use of IBM's Information Management System/360 (IMS/360) data management package.

IMS/360

The IMS/360 software package includes a comprehensive data management system called DL/I (Data Language I). This system is either being used or carefully considered by users in many large IBM installations for data base applications. One important feature of IMS/360 is that it also includes a teleprocessing

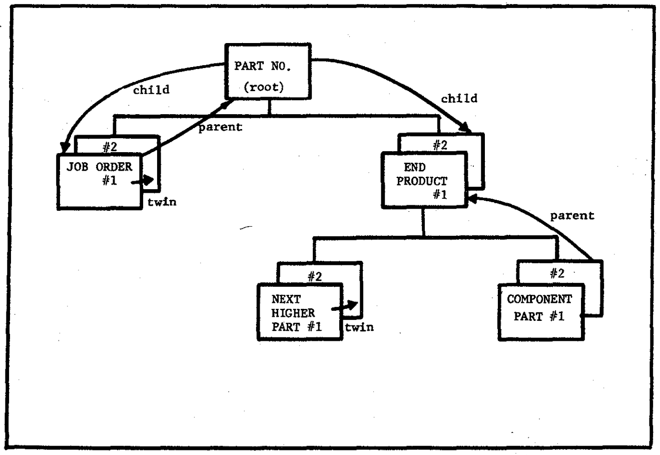


Figure 1—Data base example #1

capability which is intended to facilitate the conversion of initial batch data base applications to an on-line mode. Like most data management systems, IMS enables the user to separate the data base description from the applications programs, thus permitting certain changes to be made to the files without affecting all programs. The system also includes backup and recovery modules as well as file reorganization and statistics collecting utilities. It is clear that for most installations developing large integrated data base applications a generalized data management capability similar in scope to IMS will be required.

In addition, IMS provides a quite general structure with which to describe the data base record (or in IMS terminology, segment) relationships. This structure is a hierarchy of fixed length segments emanating from a root segment for each data base. Figures 1 and 2 both show examples of such structures. Up to 255 segment types, arranged in a maximum of 15 levels, may be specified for an IMS data base. Each segment type may occur any number of times or not at all under a given root. As shown in Figure 1, the immediate subordinate segments are referred to as child segments, the segment they appear under is referred to as the parent, and different occurrences of the same segment type are referred to as twins.

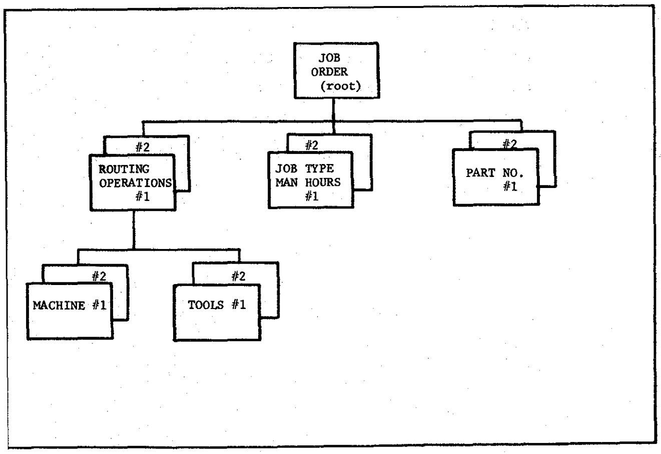


Figure 2—Data base example #2

One dominant feature of the IMS data management scheme is that the applications programmer always views the data structure as hierarchical, regardless of the access method employed, or physical location of the data. Thus the four IMS access methods all begin with "Hierarchical"; namely: the Hierarchical Sequential Access Method (HSAM), the Hierarchical Indexed Sequential Access Method (HISAM), the Hierarchical Indexed Direct Access Method (HIDAM), and the Hierarchical Direct Access Method (HDAM). When using either HSAM or HISAM, the hierarchy relationships are maintained by physically recording the segments sequentially in a top-down, left to right convention. HISAM is conceptually similar to the indexed sequential access method under O/S. It provides an index to the root segment, and a separate overflow area, as the example in Figure 3 indicates. This figure shows how a sample record from the data base in Figure 1 would be physically stored using HISAM. As many of the segments occurring under a root as can fit into a fixed length block are stored there, while the rest are chained together in other overflow blocks. The segments under a root must be accessed sequentially, and insertion of a new segment causes others to be shifted down. This necessitates periodic reorganization.

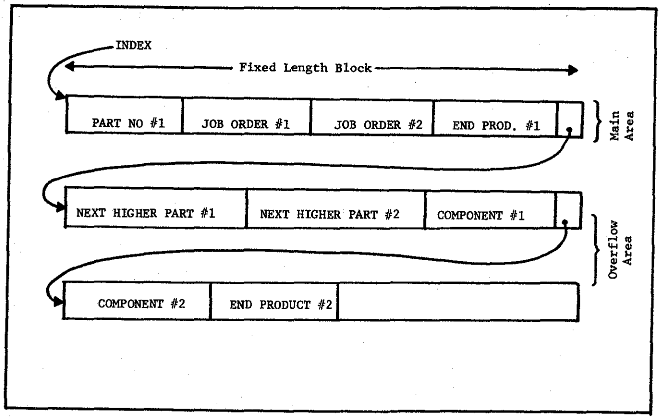


Figure 3—HISAM example

HIDAM and HDAM do not record the segments under a root sequentially, but rather allow direct pointers to the children, from the children to the parents

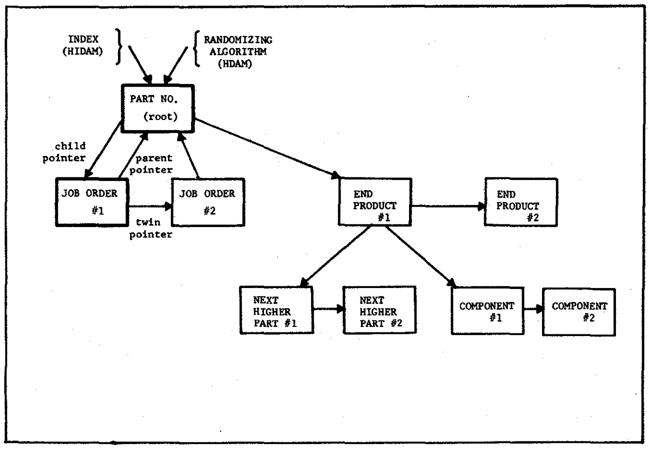


Figure 4—HIDAM and HDAM example

and among the twins under a given root, as shown in Figure 4. In effect, each segment type becomes a file. As the names imply, HIDAM provides an index to the root segments, and HDAM accesses the roots with a user supplied randomizing module.

The ability to specify relationships across data bases is provided by defining a logical data base which is composed of segments from one or more physical data bases. The sample data base assumes there is a requirement to process both parts and job orders as separate entries, but often we need to access one file from the other as well. While this can be done by repeating job orders for each part, and part numbers for each job, redundancy of data results and the programmer must perform a double maintenance task. As an alternative, IMS permits the specifying of a direct access pointer from one segment to another, as shown in Figure 5.* Here, the job order segment in the Part Data Base does not contain the job order number but rather a direct pointer to the root segment for that order number in the Job Order Data Base. Other information, called intersection data, may be recorded in the job order segment in the Part Data Base. While a similar reverse pointer can be made from the part segment under the job, an equivalent capability would be to begin a chain from the job order root, connecting all the segments for this given job order number under the various parts in the Part Data Base. To do this a "virtual" segment (shown in dotted lines) is defined. Once these direct pointers are in place, logical views of the combined data base may be specified. A logical view is a hierarchy of segments which, while not physically related in that hierarchy, can be made to appear so by utilizing the direct pointers. For example, when a programmer accesses the job order segment under a part number,

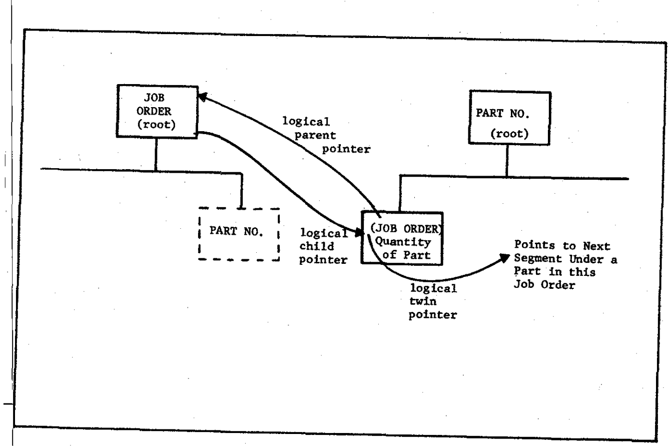


Figure 5—Use of logical pointers

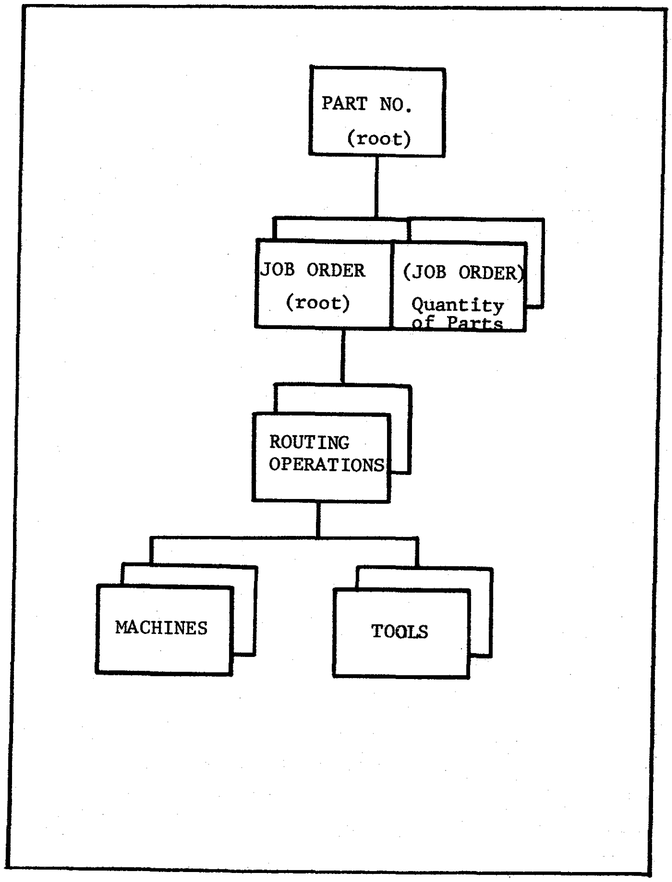
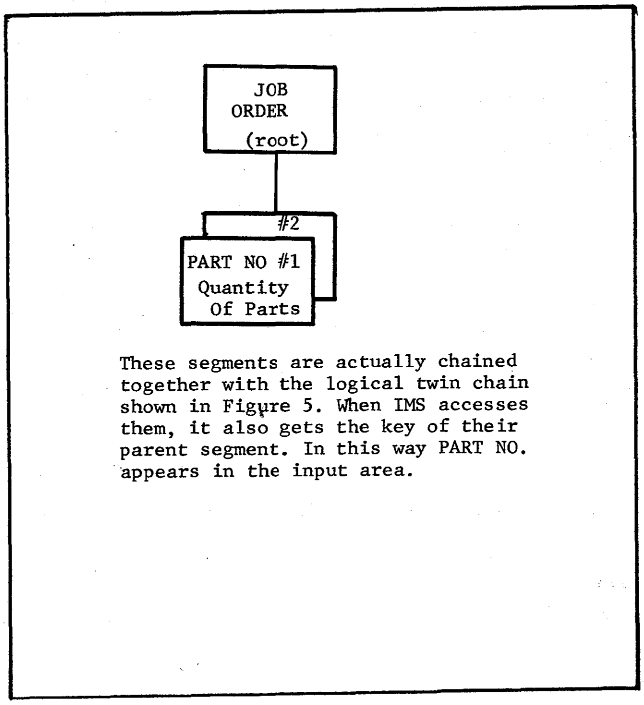


Figure 6—Logical view using logical parent

* These are referred to as logical pointers to distinguish them from the physical pointers used in the primary hierarchy as shown in Figure 4.



These segments are actually chained together with the logical twin chain shown in Figure 5. When IMS accesses them, it also gets the key of their parent segment. In this way PART NO. appears in the input area.

Figure 7—Logical view using logical child and logical twin

the record to appear in the buffer includes the job order root segment in the Job Order Data Base. Another possible logical view is shown in Figure 6.

Using other pointers, a different logical view would appear to the programmer as shown in Figure 7. Here the virtual segment appears to be under the job order root. Since IMS retrieves the key of the parent segment of a logical child segment, the part number appears in this virtual segment as well.

This brief introduction to IMS contains obvious oversimplifications, and the reader is cautioned to refer to the IMS manuals for further detail. Other features of IMS are introduced below as required.

THE DATA BASE DESIGNER

The use of a generalized data base management system, and the desire for integrated data bases, both necessitate centralization of the file design effort, rather than distributing it among the various application programmers, for example. Much has been written about the importance of the data base designer and his role as an interface among the applications teams. Experience with IMS reinforces this view. Moreover, experience indicates that the data base designer must be knowledgeable in the applications at hand. The data base design permeates the applications to such a degree that if any hope of efficiency is to be realized, the data base designer must be able to make positive contributions to program and job stream flow based on optimizing data base performances (relative to the time and/or storage measures as discussed above). Thus, it may be *easiest* from the application programmer's point of view to generate and deal with a particular logical structure, say as shown in Figure 7. But it is the data base designer who knows that each part record under the job number will require at least two physical accesses; he can suggest the duplication of the part number as a trade-off possibility.

Another reason that the data base designer must be familiar with the applications is that application dependent features are actually coded in the data base. For example, IMS permits the optional specification of only unique keys for multiple occurrences of a given segment type. An attempt to add a duplicate key will cause a certain message to be returned and this may be significant in the program logic. Another instance of such dependency concerns the addition and deletion of segments using logical relationships. IMS permits several options with regard to adding or deleting from a logical view. In Figure 7 for example, suppose a program reads a job number, and then deletes a part under this job number. Certain IMS coding may now cause the part number in the Part Data Base to be deleted as well. Obviously, this coding should only be specified after a thorough understanding of the application.

UTILIZATION OF RESOURCES

The price to be paid in the use of a generalized data base management system is some overhead in resource utilization, typically storage space, CPU time, or accesses. It is a mistake to assume this overhead is fixed, and invariant to the data base design. Quite the opposite is true: since each task consumes more resources, the opportunity (and in many cases the necessity) for efficiencies is very prevalent. A methodology of data base design then, depends largely on estimating the "overhead" or cost of certain options, in order that profitable trade-offs can be made. Unfortunately, as with many other systems, the costs associated with various IMS features are not publicized, and in some instances can even be counter-intuitive. In most cases the data base designer must extrapolate from his knowledge about the internal workings of the data management system in order to estimate overhead.

Another input to design trade-off studies is data base statistics. Accurate statistics about the data are vital to an efficient data base design. Note that in some instances, however, the use of a data management system

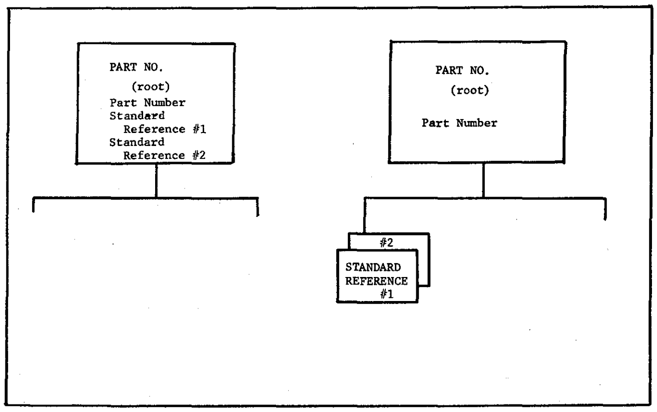


Figure 8—Creating a new segment type

removes the necessity of obtaining accurate statistics prior to data base design. These instances involve precisely the parameters which we are allowed to alter without affecting the applications programs, since these can be changed easily after actual experience with a loaded file has been achieved. The use of IMS permits the data base designer to allocate different physical space to a file, change between HDAM and HIDAM, add a new segment to certain places in the data base, or change blocking factors, all without affecting an application program. But he may not alter fields within a segment, add a new segment to certain places, or modify the logical pointers without incurring some rewriting of the programs. To the degree that the data base is more or less fixed, the trade-offs will only be as good as the accuracy of the statistics upon which they are based.

To illustrate the preceding points, consider this trade-off problem concerning the specification of new segment types. Basically the question is "under what circumstances is it best to create a new segment type?" One case arises when subordinate data may repeat a number of times; it is clear that a new, repeatable, segment is preferable to a large space reserved for the maximum data possible. But what if we know the data can only repeat twice for example? More specifically, suppose we wish to record references to standards for each part, up to a maximum of two standards per part. The trade-off is then between allowing for two such fields within the part root segment, or creating a new segment type subordinate to the part root segment containing a standard reference, as depicted in Figure 8. What factors should be taken into account in making this trade-off? The factors include the following items:

- The IMS storage overhead associated with each new segment occurrence;
- The added complexity of the data base description;
- The unused space if a field is always present but has no value;
- If HIDAM or HDAM are used, the space for the pointer to a subordinate segment from the parent, and the pointer connecting the child twins;
- The accesses necessary to obtain the data in a different segment;
- If HISAM is used, the time to process each new segment type after the block is in core.

Thus the trade-off among storage, accesses, and CPU time must address these factors. To simplify things though, assume we merely wish to minimize storage space, and each standard reference consumes 8 bytes. Each IMS pointer requires 4 bytes, and there is a 4 byte overhead for each segment occurrence. If we allow for 2 standards in the part root we clearly require 16 bytes per part no matter what. But the storage requirements for a new segment type depend on the actual distribution statistics of standards per part. If very few parts have standards then we are better off with a new segment type. Clearly also, if most parts actually do have two standards then one segment is best since each new segment requires at least 12 bytes, 4 for overhead, and 8 for data (if HIDAM or HISAM are used 4 more bytes for twin pointers plus 4 bytes for a pointer from the part root to the new child are required as well). If the actual occurrences lie somewhere in between then more accurate statistics are probably needed for the trade-off to be made. Otherwise, if either method is likely to result in about the same storage requirement, then some other factor, probably number of accesses, would be optimized.

DATA BASE MAINTENANCE

Whereas on-line inquiry or status posting applications are the more interesting ones, it is very often the batch file update and reorganization runs which take up the vast majority of system resources. The file designer must be sensitive to the batch update requirements because these can become system bottlenecks just as easily as the on-line applications. In attempting to optimize the overall system, a very delicate trade-off decision is required.

Again, good statistics make for an informed decision. An especially important use of these statistics is to estimate file sizes and growth. File size estimates are needed since the size of the file will directly affect the time required to backup and restore the file or to reorganize it. This is in addition to the input of file size estimates to hardware configuration planning.

When using a data base management system such as IMS/360, the data base designer must take into account the storage requirement imposed by the system, including pointers, control fields, and indices. Actual experience has shown that storage overhead for pointers and IMS control fields can easily reach 50 percent of the total storage requirement. In applications approaching a billion bytes, this overhead becomes a very costly factor. Not only must the cost for physical storage be borne, but the maintenance load is proportionally increased, resulting in a greater processing requirement. Thus the trade-off between storage and accesses should only be made considering the entire system—batch data base maintenance as well as on-line transaction processing.

A final note then about accesses. Hidden accesses in IMS (i.e., the *average* ratio of logical accesses to physical accesses) has run as high as 4 or 5 to 1. Translating into accesses per transaction, the result often shows that between 20 and 50 physical accesses are required per transaction. More complex transactions such as a Bill of Materials update can require hundreds of accesses per item. One can see here that a data base design which minimizes hidden accesses can affect a reduction in running time of a B/M processor by substantial margins.

CONCLUSIONS

One can argue that a data base management system should not be chosen until the optimal file structure has been identified. In this way a system which supports that structure can be selected—rather than forcing an application into a structure dictated by the system and thereby paying in performance. While this argument has merit, practical considerations often leave no choices open. To some degree this will be the case with many IMS/360 users. It is the only data base management system supported by IBM for large applications. It is one of the few systems now supporting TP. It has many (but not all) of the backup and recovery features needed for large data base applications. Other systems like the Honeywell Integrated Data Store offer file structuring capabilities which may be more suitable to a particular application, but are not implemented on IBM hardware.

While this may sound fatalistic, it points up the need to be especially careful in designing files under these circumstances. Too many users have the view that the use of IMS/360 or any other similar system precludes him from paying much attention to file design—that the system will design the files. The examples above show that this is not the case. The user should view the data base management system as a tool in implementing a design which has been arrived at by taking into account both the applications requirements and the features and limitations of the generalized system. Only in this way can he expect both reasonable performance and overhead.

REFERENCES

1 F H BENNER
On designing generalized file records for management information systems
AFIPS Conference Proceedings Vol 31 1967 Fall Joint Computer Conference

2 N CHAPIN
A comparison of file organization techniques
Proceedings of the ACM 24th National Conference
San Francisco California August 1969

3 A M COLLMEYER
File organization techniques
IEEE Computer Group News March/April 1970

4 A M COLLMEYER J E SHEMER
Analysis of retrieval performance for selected file organization techniques
AFIPS Conference Proceedings Vol 37 1970 Fall Joint Computer Conference

5 G G DODD
Elements of data management systems
Computing Surveys Vol 1 No 2 June 1969

6 J K LYON
An introduction to data base design
John Wiley & Sons Inc 1971

An information structure for data base and device independent report generation

by C. DANA and L. PRESSER

University of California[*]
Santa Barbara, California

INTRODUCTION

The generation of computer output information that is easily read by humans is a tedious and elaborate task when present-day programming languages are employed. This is certainly true when assembly language is used. Features like PL/I's *DATA* and *LIST* output options are improvements over FORTRAN's requirements of mandatory *FORMAT* statements. However, such desirable formats as tabular listings still require much programming and coordination. Another troublesome area involves the generation of information (i.e., reports) on devices that have different characteristics (e.g., printer, CRT). Typically, the size of the "page", and thus the amount of information that can be placed on a "page" will be different for the various devices. Current programming practice forces specification of the output format in a device dependent manner. Thus, to generate a report on more than one type of device would require recoding the section of a program that specifies the output format. Therefore, a method for describing just the logical format of a report, without consideration of the characteristics of the possible output device, would be desirable.

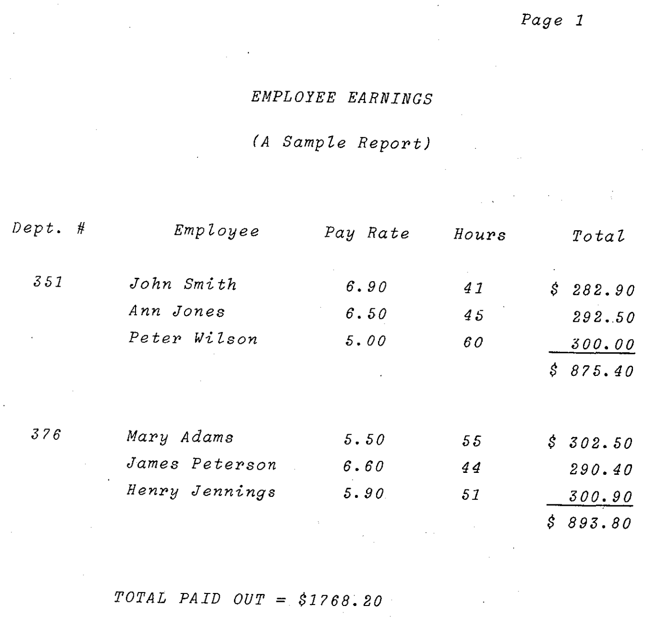
Once a device independent description of a report is obtained, it is a natural extension to attempt to separate the specification of the report from any given file or data base. Consequently, a generalized information structure for data base and device independent report generation is obtained. To generate reports it is necessary to implement a system that, with the device specification as a parameter, interacts with the information structure and the data base in order to generate actual output.

This paper describes a data base and device independent information structure for the representation of reports, the environment in which the structure resides, and the support programs that allow the generation of output.

It should be noted that a report need not be limited to business applications as is now generally thought. All areas of computer applications can benefit from uncomplicated output coding and from an orderly presentation of information. Indeed, there is a set of "output needs" that is common to most application areas. For example, observe the structural similarity between a report on employee's earnings (Figure 1) and a report on the performance of subroutines (Figure 2). It is our firm opinion that every (computer) system should incorporate at design time facilities for debugging and measurement purposes. Hence, the need to output information is inherent and ubiquitous, and thus, should be an integral consideration in the design of any programming language.

It is worthwhile at this point to discuss the general characteristics of a report. By a *report* we imply any visual computer output that is easily understood by humans. A report is not a static entity. That is, one can not, in general, lay down on a piece of paper the exact line and column positions of all items that will be part of a page and then complete the final report by just filling in the assigned areas with values. There can be sections of a report that may be repeated a number of times, the actual number being known at the time values are read from a data base. In Figure 1 each line in the body of the report corresponds to one employee, and the number of employees can vary from month to month. Similarly, in Figure 2, the number and frequency of subroutines used can vary from execution to execution. Therefore, since the final form of a report is not known until the data base is read, the logical description of a report must specify how input records are to be obtained, processed, and actual output generated. In general, one could conceive of a report whose final

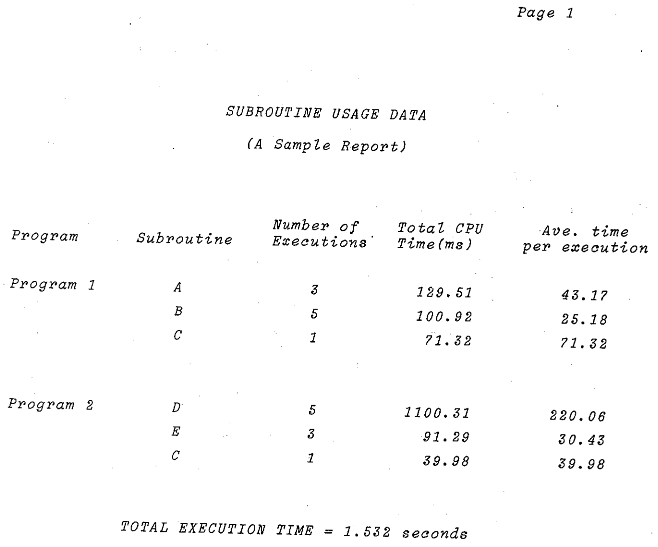
[*] Department of Electrical Engineering. This work was supported in part by the National Science Foundation, Grant GJ-31949.



```
                                                     Page 1



                         EMPLOYEE EARNINGS

                         (A Sample Report)


Dept. #        Employee           Pay Rate    Hours       Total

  351        John Smith             6.90       41      $ 282.90
             Ann Jones              6.50       45        292.50
             Peter Wilson           5.00       60        300.00
                                                       $ 875.40


  376        Mary Adams             5.50       55      $ 302.50
             James Peterson         6.60       44        290.40
             Henry Jennings         5.90       51        300.90
                                                       $ 893.80


             TOTAL PAID OUT = $1768.20
```

Figure 1—Employee earnings report

form would depend on a wide variety of relationships among variables. For instance, in our business example, we may wish to flag the names of those employees who have worked more than a fixed number of hours and whose pay rate is above a certain level. Similarly, in the measurements example, we may wish to flag the names of those subroutines that executed more than a fixed number of times and required more than a certain period of time. Thus, a facility for testing relationships between variables and performing actions based on the results is needed for the logical description of reports.



```
                                                    Page 1



                      SUBROUTINE USAGE DATA

                        (A Sample Report)


                          Number of    Total CPU      Ave. time
Program      Subroutine   Executions   Time(ms)     per execution

Program 1       A             3         129.51          43.17
                B             5         100.92          25.18
                C             1          71.32          71.32


Program 2       D             5        1100.31         220.06
                E             3          91.29          30.43
                C             1          39.98          39.98


          TOTAL EXECUTION TIME = 1.532 seconds
```

Figure 2—Measurements report

The data base may not include all the values that are to appear in a report. For instance, we may wish to output total pay as part of a report when only pay rate and hours worked is present in the data base. Or we may desire to output average subroutine execution time when only total execution time and number of calls is present in the data base. Therefore, a facility to carry out calculations is needed for the logical description of reports.

In summary, the mechanisms needed for the logical specification of reports are the basic elements of a programming language. In fact, the information structure described here can be viewed as a special purpose report generating machine. It is also possible to view it as an intermediate representation for the translation of the output sections of programs. Indeed, it is this latter line of thought that motivated this work.

INFORMATION STRUCTURE ENVIRONMENT

Our environment for report generation is outlined in Figure 3. The user specifies, in some report generator language, the form of the desired report. For our purposes a *report generator language* is any language that possesses the facilities needed to describe the desired reporting. It may be a language designed specially for report generation (e.g., RPG), a more general language that includes special facilities for report generation (e.g., COBOL), or it may be a general purpose language (e.g., PL/I, assembly language). In the latter two cases the report generation code may only be part of a larger program.

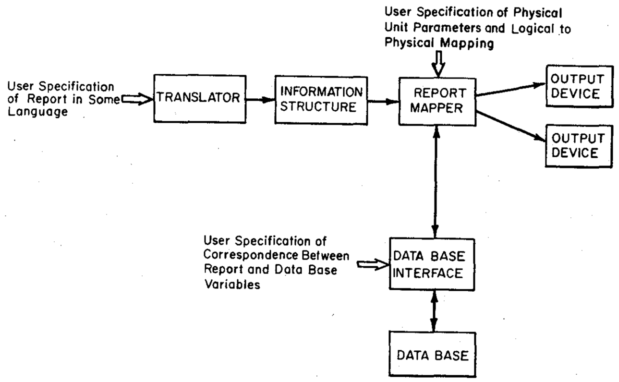
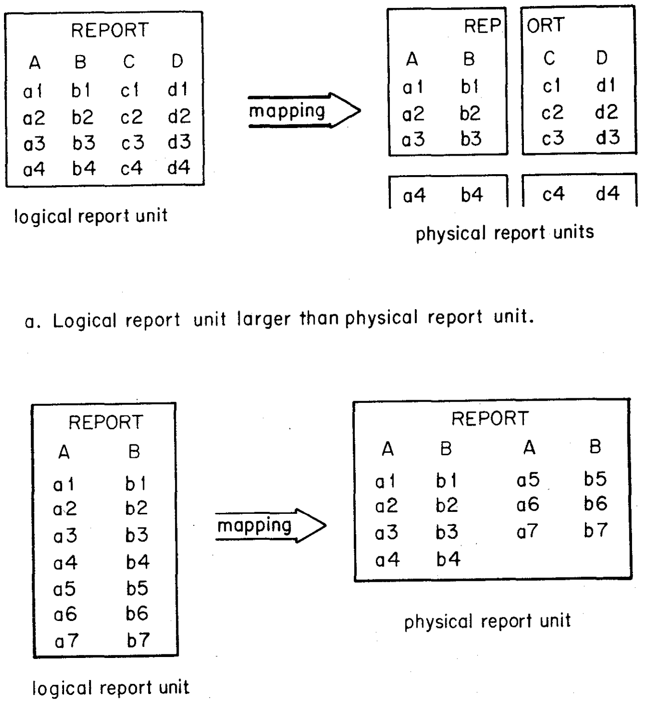

User Specification of Physical Unit Parameters and Logical to Physical Mapping
User Specification of Report in Some Language
TRANSLATOR
INFORMATION STRUCTURE
REPORT MAPPER
OUTPUT DEVICE
OUTPUT DEVICE
User Specification of Correspondence Between Report and Data Base Variables
DATA BASE INTERFACE
DATA BASE

Figure 3—Information structure environment



a. Logical report unit larger than physical report unit.

b. Logical report unit narrower than physical report unit.

Figure 4—Sample mappings

The user's program is translated such that the logical description of the report is mapped into the information structure. (This information structure is discussed in detail in a later section of this paper.) The key information about the report, as originally described by the source language and now embodied in the information structure, is called the *logical report.* This is a specification of the report (e.g., where the headings are to be and what they are to state) in terms of some virtual (nominal size) surface called the *logical report unit.* The logical report unit consists of a fixed number of rows and columns. When the report finally appears on an output device media (e.g., paper, CRT face) it is called the *physical report* and the size of the actual display surface of the device is termed the *physical report unit.*

The mapping of a logical report unit into one or more physical report units is called the *logical to physical mapping* or simply the *report mapping.* Such a mapping includes obtaining data from the data base and the proper placing of results in the logical report, before generating a physical report unit. Report mapping is carried out by a program referred to as *report mapper* in Figure 3. Based on the information structure, the mapper fills out the logical report unit and then employs default or user specified parameters in order to carry out a logical to physical mapping. Examples of possible mappings are shown in Figure 4.

In general, soft-copy devices require mappings different from those employed with hard-copy units. In the case of a hard-copy device the logical report units can be split at arbitrary places to satisfy the physical report unit. The hard-copy segments can be later placed side by side and viewed as a whole. On the other hand, in the case of a soft-copy device the output can only be viewed one physical unit at a time; thus, care must be exercised to make each display coherent and readable. For example, the mapping shown in Figure 4a would not be very meaningful if the physical units must be viewed separately. In such a situation it is necessary to repeat identifying information and to make sure that all items are properly placed. Specific mapping algorithms are beyond the scope of this paper.

The report mapper buffers the physical report units before sending these units to the output device. The buffer size is important. If it is smaller than the logical report unit, placement of information would be restricted. For instance, a total could not be placed at the top of a physical report if the physical report unit that corresponded to the top of the physical report had already been sent to the device by the time the total was obtained.

In order to complete the discussion of the environment outlined in Figure 3 we need to describe the interface with the data base. The user must specify the correspondence between the variables present in the information structure and those residing in the data base. The function of the *data base interface* module (refer to Figure 3) is to supply the report mapper program with any data needed to generate a report.

INFORMATION STRUCTURE

The information structure consists of a number of tables (lists) each of which describes a section of the report format or generation process. Entries in each of the tables contain pointers to entries in other tables, thus, a linked structure is formed as depicted in Figure 5.

The *Report Head* describes the gross structure of the report. It contains the dimensions of the logical and physical report units. The latter may be supplied by the user or may be set, by default, to the value of the logical report unit. The report head also specifies any actions to be carried out at the beginning/end of the report and at the beginning/end of each logical page. The *report body action* entry is responsible for all of the other details of the report and, in essence, it represents the bulk of the reporting activity. The action entries in the *Report Head* point to a list of actions in the *Action Table.*

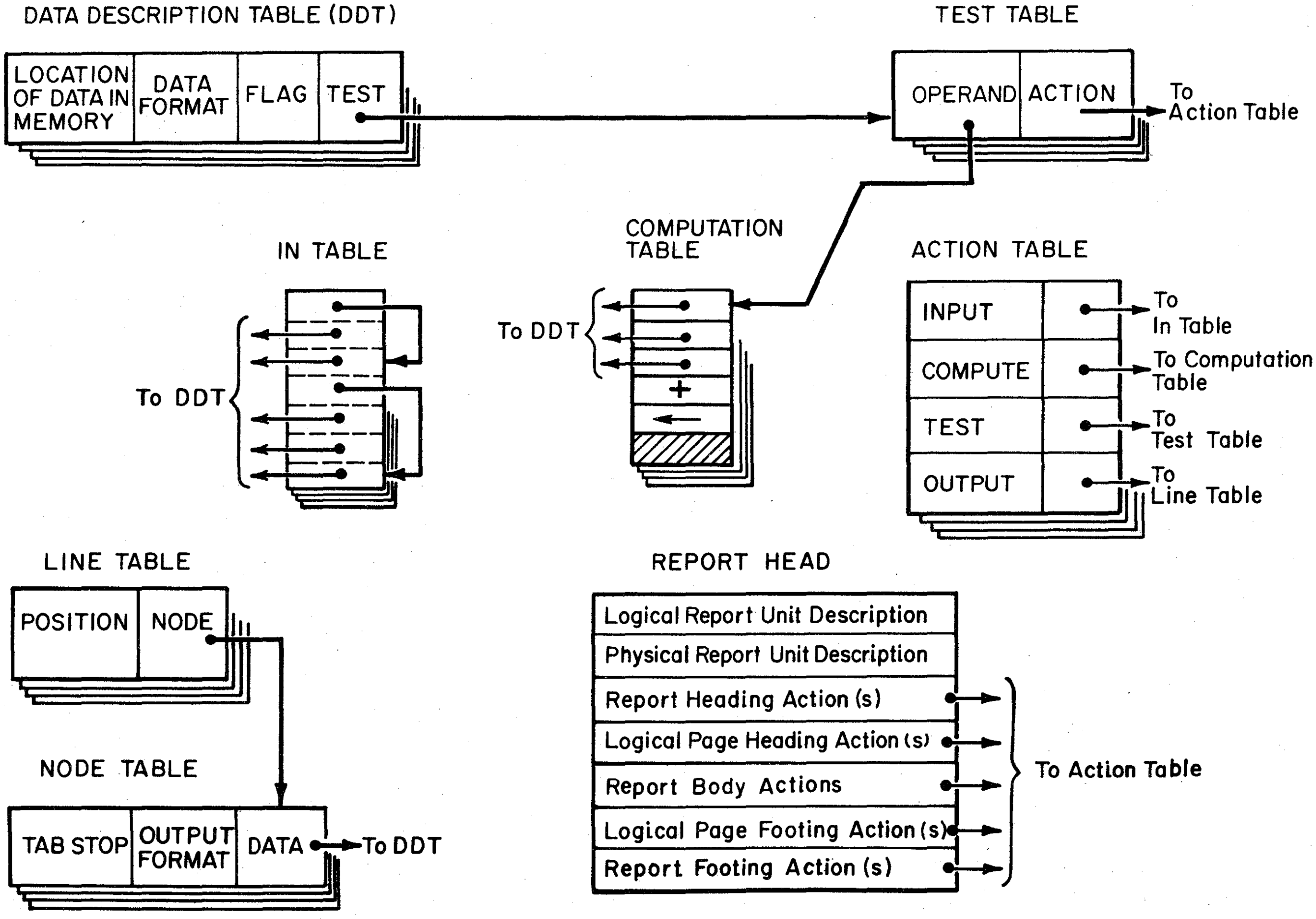


Figure 5—Information structure

The *Action Table* lists the sequence of actions that comprise the report generating process. There are four types of actions: *input* actions to obtain data from the data base; *compute* actions (including logical operations); *test* actions to determine flow of control; and output actions to create actual output. The detailed specification of these actions is contained in the *In Table, Computation Table, Test Table,* and *Line-Node Tables* respectively. Flow of control in the *Action Table* is sequential unless a transfer occurs as a result of a test.

The *Data Description Table* (*DDT*) contains information about the *location* and *format* of the data elements manipulated in the report. All references to data in any other table is specified by a pointer to a *DDT* entry. The *DDT* entry in turn contains a pointer to the location in memory where the actual datum is stored. There are two other fields in each *DDT* entry. The *flag* field is used to represent one of three possible conditions: (1) there are more data values to be input from the data base; (2) there is not more data available from the data base (i.e., "end of file"); (3) this datum represents an internal variable.* The *test* field specifies any test that is to be performed when the datum receives a new value. In essence, this facility implements an "on-condition". Such a capability may be exploited in the report generator language to free the user from having to specify a detailed ordering of calculations.

The *In Table* consists of a set of nodes, where each node is associated with an *input* action in the *Action Table.* A node consists of an ordered list of pointers to the *DDT*; the first entry of a node points to the last entry. The *DDT* pointers pinpoint which data values are to be input from the data base.

The *Computation Table* is a linear list. This table contains a postfix (Reverse Polish) representation of the computations to be performed on data. Each entry rep-

* Internal variables are those created to store intermediate values during report generation. It is assumed that a segment of memory is dedicated to auxiliary storage.

resents: an operand (i.e., pointer to the *DDT*), an operator, or an end of computation marker. The sequences of computations corresponding to *compute* actions in the *Action Table* are delimited by markers.

The *Test Table* contains the specification of the tests to be carried out and the action to be taken if the end result is true. The *operand* field points to the postfix representation of the test. The *action* field points to the action to be executed if the result is true. Note that tests are activated at two possible times: after manipulating (e.g., input operation) a *DDT* entry if the *test* field of the *DDT* entry is not null; or when control flows into a *test* in the *Action Table.*

The *Line Table* and the *Node Table* together specify the format of rows of the logical report described by the information structure. These tables support *output* actions. The *Line Table* defines line (row) position information for the lines of the logical report units. A line may have either a relative or absolute position specified in the *position* field. The relative position relates to the previous line and is employed with those sections of the report that may be repeated an indefinite number of times. For example, referring to Figure 1, relative positioning would be used to output a summary of employees' earnings when the number of employees to be reported is not known until the data base is read. An absolute line position corresponds to a fixed distance from the top of the logical report unit. It is used for those sections of the report whose positions will not vary from (logical report) unit to unit: for instance, page numbering. The segments of a line of output are defined by a list of nodes stored in the *Node Table.* The first node is specified by the *node* field in the *Line Table.* Each entry in the *Node Table* specifies the tab (column) position of the corresponding line segment, in the *tab stop* field; the external format in which data is to be output, in the *output format* field; and a pointer to a *DDT* entry for the data item to be output, in the *data field.* The *tab stop* field may also indicate a relative or absolute position.

Next, we discuss the support programs that allow the actual generation of output.

IMPLEMENTATION

The implementation of the system we have described is divided into two main parts. First, it is necessary to have a *Translator* that transforms a user's specification of a report into our information structure form. Such a transformation is not much different from that carried out by conventional translators; thus, it will not be discussed here. For an overview of the subject see Reference 1. The second part of the system is the *Report Mapper.* The basic design of this unit consists of a series of essentially independent subprograms. In terms of Figure 5, each subprogram relates to a particular table, and each performs the functions associated with the table in question. The subprograms employed in an experimental implementation are tabulated in Table I.

TABLE I—Subprograms in Report Mapper

Subprogram name	Associated Table	Function
ACTION	ACTION	Causes other subprograms to be called as directed by the Action Table.
COMPUTE	COMPUTATION	Performs the operation specified in the computation table.
DATA	DATA DESCRIPTION	Accesses information through DDT and passes it to other subprograms.
LINE	LINE	Causes a line of the report to be properly positioned and triggers beginning/end of page actions.
NODE	NODE	Converts internal data format to external format and positions data in report buffer.
TEST	TEST	Causes execution of logical test and subsequent transfer of control.
INPUT	IN	Obtains new value for input variable(s).

As an illustration, let us examine how the *Node* subprogram operates in order to place a segment of a logical report line in a physical report. This subprogram is called by the *Line* subprogram and it is passed the address of the node describing the first segment of the line to be output. *Node* obtains a pointer to the *DDT* from the first entry in the *Node Table.* Next, with the aid of the *Data* subprogram, the desired datum is obtained as well as a description of its format. Then, the *data* and *output format* specifications are compared, and if they differ, a conversion to the otuput format specified by the *Node Table* entry is effected. Finally, the converted datum is positioned in a physical report buffer, as specified by the *tab stop* field of the *Node Table* entry and the report mapping algorithm. The remaining segments of the logical report line under consideration are processed in a similar fashion. The physical report is composed unit by unit in a buffer. When the buffer is full it is output by a device dependent routine and reset to blanks.

It is worthwhile to observe that the implementation described here mechanizes a pseudo-machine whose instruction repertoire consists, in essence, of the four actions: *input*, *compute*, *test*, and *output*.

SUMMARY

In this paper we have presented an information structure for report generation that separates the data base and device dependent parts from the logical description of a report. Such a representation of reports (i.e., output) allows a clean and elegant interface to data bases and devices. It also brings out with some strength the need for better output facilities in current programming languages. Furthermore, it may serve as a guide in the design of report generator language facilities.

REFERENCE

1 L PRESSER
The translation of programming languages
In *Computer Science* A Cardenas L Presser and M Marin (Eds) John Wiley and Sons Inc New York 1972

SIMS—An integrated, user-oriented information system

by M. E. ELLIS, W. KATKE, J. OLSON, and S. C. YANG

University of Wisconsin
Madison, Wisconsin

INTRODUCTION

SIMS, a Social Science Information Management System, has been developed by the Social Science Data and Computation Center (DACC) at the University of Wisconsin for the purpose of providing the social scientist with a personalized data base management system (DBMS). Over the past four years many DBMS were surveyed and their basic features compared to our requirements. All systems surveyed had their strong and weak points but no one system was found to satisfy all our needs. In addition, new and improved methods and algorithms for language processing, storage and retrieval, and analysis capabilities have been developed by DACC[1,2,3,4,5] and others since the advent of these earlier systems. These facts, coupled with the fact that conversion of existing systems seemed impractical due to their machine dependence, has resulted in the development of SIMS. It is the intent of SIMS through its features and basic design to serve as an information processing tool for social scientists and others working with large and complex data files.

FEATURES

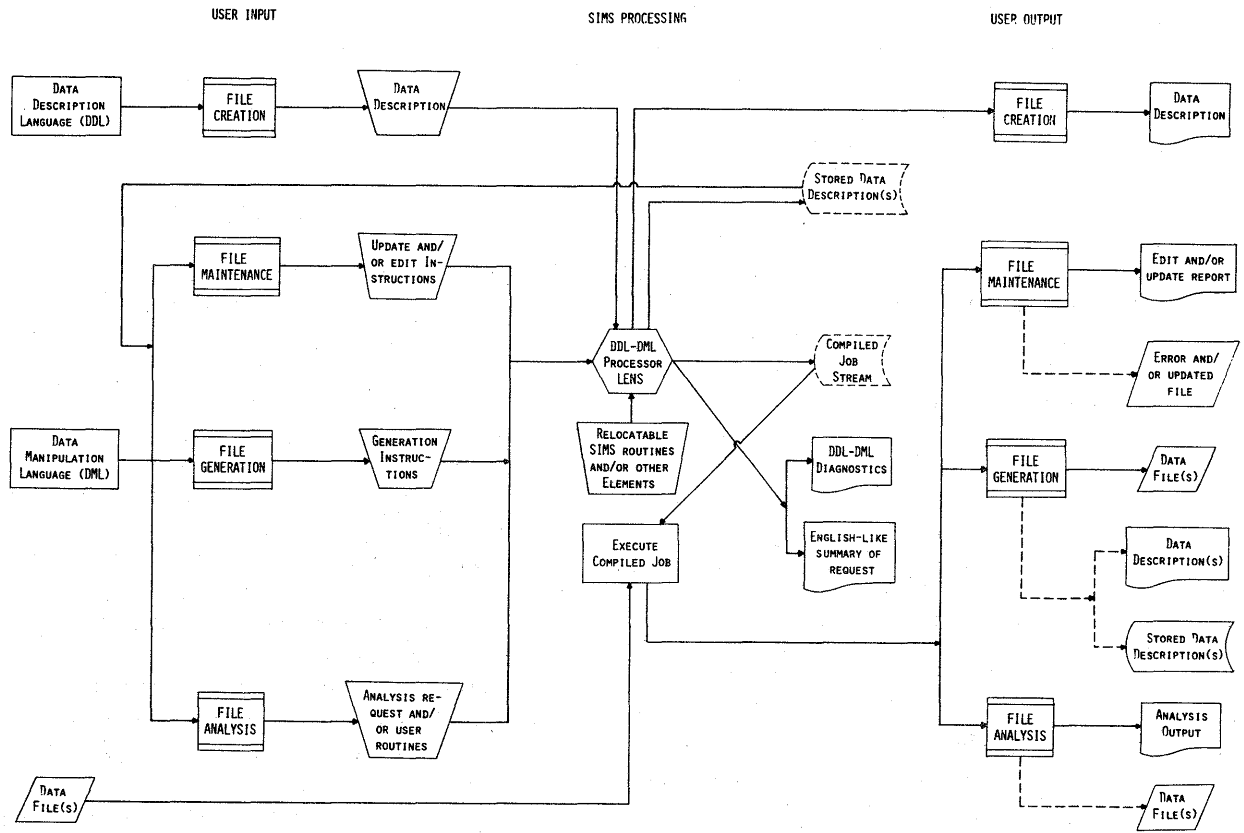
In establishing the requirements for SIMS we were cautious to heed Sibley's warning to the effect that "the extent to which we set sights too high or inconsistent goals, especially among users, we do a great disservice to the industry, both users and suppliers."[6] Requirements were gleaned from past experience and continuously updated as the system developed. These requirements have evolved into the present features of the SIMS system, which are briefly as follows:

- SIMS provides a complete repertoire of integrated processing functions for the creation of machine readable data files; data validation, consistency checking, maintenance and retrieval; and statistical and other analytical processing of data.
- SIMS provides the user with both on-line and batch communication with the system. The syntax of the SIMS language is conducive to allowing the user to prepare his requests on cards or precoded forms for batch submittal, or enter them on-line in free field format from a teletypewriter. These means of user interaction satisfy the requirements of both an experienced and non-experienced SIMS user.
- SIMS enables non-programmers to use the system via the user-oriented, descriptive request language and at the same time allows programmer types the latitude of combining their Fortran routines with the system. Output options and formats for most processing functions are predetermined but can be annulled or supplemented via user supplied routines.
- SIMS is capable of processing hierarchical files of *n* levels composed of fixed length data items. Such files are characteristic of most quantitative data processed on computers.
- SIMS has a complete data definition capability which allows the user to describe his data meaningfully using terms familiar to him, and to retain the description in machine readable form for easy updating and use by the system.[7] This feature applies to data already in machine readable form (such as information produced on another computer) or data which is coded by the user of SIMS. Data descriptions of previously defined data files may be automatically modified or created by the system as a result of a processing function, such as a merge or extract, or may be altered by the user. This feature guarantees continual file documentation and provides a redefinition capability and an efficient retrieval of information based on the individual's specific use of a particular file or files.
- SIMS provides three modes of operation, DIAG-

NOSTIC, TEST and PRODUCTION. DIAGNOSTIC mode checks the completeness and accuracy of the input request. TEST mode extends the job past the DIAGNOSTIC stage to include processing of the first *n* entries of a file for each function specified. PRODUCTION mode implies continued execution until job termination.

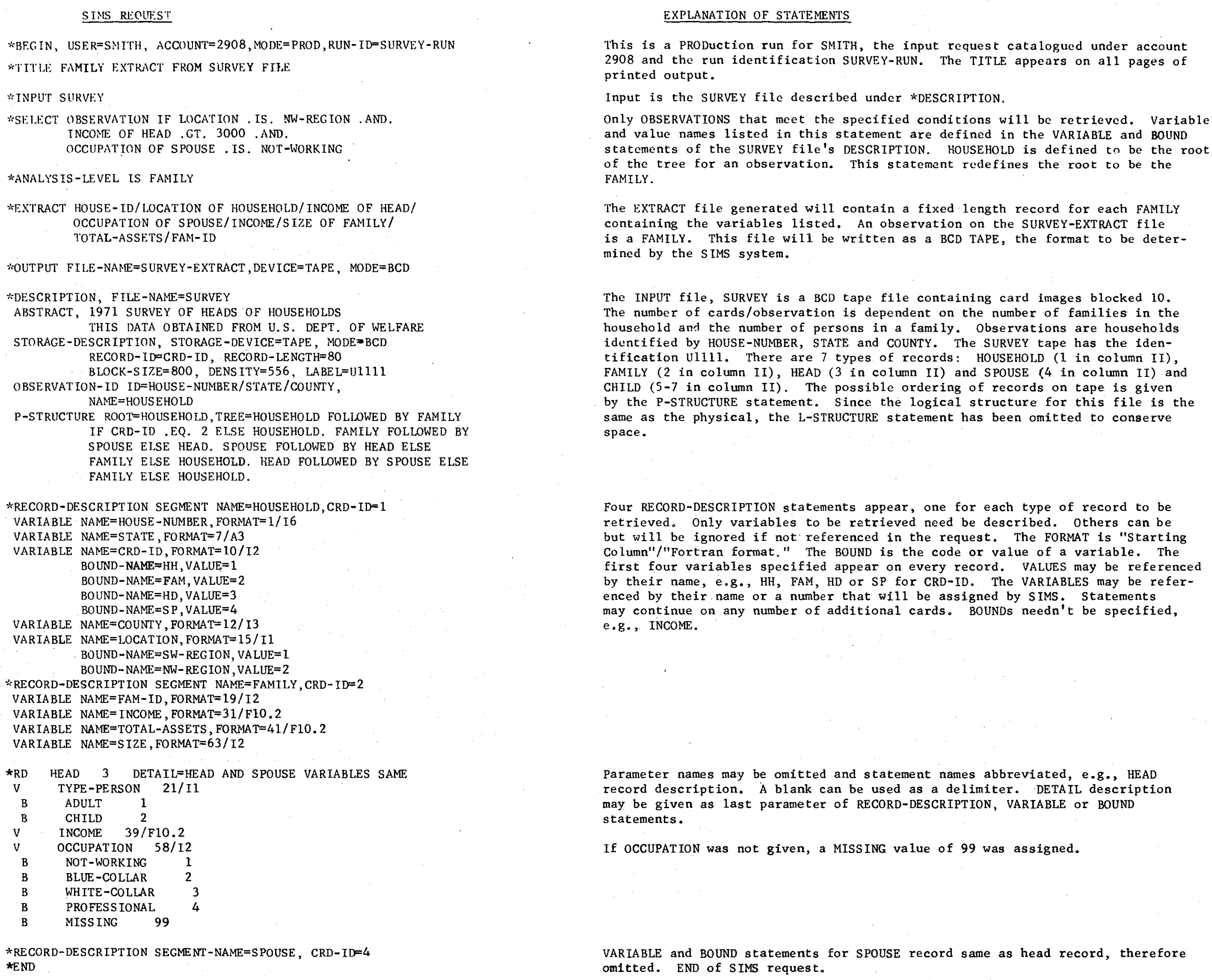
- SIMS restates the problem posed by the user in complete English sentences enabling the user to easily verify that the abbreviated SIMS statements of his request are an accurate and complete description of what is to be performed.
- SIMS through its TEACH function provides users with an interactive query facility for learning about SIMS.
- SIMS provides an option for the user to obtain certain intermediate output from the system which the user may modify to compensate for special features not present in SIMS or inefficiencies in the software. These inadequacies may occur during development or if SIMS is transported to another installation.
- SIMS has been designed, implemented and documented in a manner which lends to a minimum number of conversion problems and problems with extending or modifying the system. SIMS uses a combinatorial approach in system design and employs a network structure to communicate among subsystems. Every attempt has been made to code routines in ANSI Fortran or Cobol whenever possible. Non-standard statements are flagged as well as all Format statements, calls to installation dependent routines or any other statements which may present possible conversion problems.[8]

The features listed above describe the general characteristics of SIMS and, if viewed in detail, would reflect the system's fulfillment to its diverse scope of users. The undergraduate student using the computer for the first time represents one end of the spectrum and the programmer with multi-language familiarity the other. Included also is the "foreign" researcher who has no prior knowledge of Wisconsin's operating system control language and its idiosyncrasies, and whose alternatives



NOTE: DASHED LINES DENOTE OPTIONAL OUTPUT. DASHED BOXES DENOTE INTERMEDIATE SYSTEMS OUTPUT AVAILABLE TO USER.

Figure 1—SIMS processing functions and information flow



SIMS REQUEST

EXPLANATION OF STATEMENTS

```
*BEGIN, USER=SMITH, ACCOUNT=2908,MODE=PROD,RUN-ID=SURVEY-RUN
*TITLE FAMILY EXTRACT FROM SURVEY FILE
```

This is a PRODuction run for SMITH, the input request catalogued under account 2908 and the run identification SURVEY-RUN. The TITLE appears on all pages of printed output.

```
*INPUT SURVEY
```

Input is the SURVEY file described under *DESCRIPTION.

```
*SELECT OBSERVATION IF LOCATION .IS. NW-REGION .AND.
        INCOME OF HEAD .GT. 3000 .AND.
        OCCUPATION OF SPOUSE .IS. NOT-WORKING

*ANALYSIS-LEVEL IS FAMILY
```

Only OBSERVATIONS that meet the specified conditions will be retrieved. Variable and value names listed in this statement are defined in the VARIABLE and BOUND statements of the SURVEY file's DESCRIPTION. HOUSEHOLD is defined to be the root of the tree for an observation. This statement redefines the root to be the FAMILY.

```
*EXTRACT HOUSE-ID/LOCATION OF HOUSEHOLD/INCOME OF HEAD/
         OCCUPATION OF SPOUSE/INCOME/SIZE OF FAMILY/
         TOTAL-ASSETS/FAM-ID

*OUTPUT FILE-NAME=SURVEY-EXTRACT,DEVICE=TAPE, MODE=BCD
```

The EXTRACT file generated will contain a fixed length record for each FAMILY containing the variables listed. An observation on the SURVEY-EXTRACT file is a FAMILY. This file will be written as a BCD TAPE, the format to be determined by the SIMS system.

```
*DESCRIPTION, FILE-NAME=SURVEY
 ABSTRACT, 1971 SURVEY OF HEADS OF HOUSEHOLDS
          THIS DATA OBTAINED FROM U.S. DEPT. OF WELFARE
 STORAGE-DESCRIPTION, STORAGE-DEVICE=TAPE, MODE=BCD
          RECORD-ID=CRD-ID, RECORD-LENGTH=80
          BLOCK-SIZE=800, DENSITY=556, LABEL=U1111
 OBSERVATION-ID ID=HOUSE-NUMBER/STATE/COUNTY,
          NAME=HOUSEHOLD
 P-STRUCTURE ROOT=HOUSEHOLD,TREE=HOUSEHOLD FOLLOWED BY FAMILY
          IF CRD-ID .EQ. 2 ELSE HOUSEHOLD. FAMILY FOLLOWED BY
          SPOUSE ELSE HEAD. SPOUSE FOLLOWED BY HEAD ELSE
          FAMILY ELSE HOUSEHOLD. HEAD FOLLOWED BY SPOUSE ELSE
          FAMILY ELSE HOUSEHOLD.
```

The INPUT file, SURVEY is a BCD tape file containing card images blocked 10. The number of cards/observation is dependent on the number of families in the household and the number of persons in a family. Observations are households identified by HOUSE-NUMBER, STATE and COUNTY. The SURVEY tape has the identification U1111. There are 7 types of records: HOUSEHOLD (1 in column II), FAMILY (2 in column II), HEAD (3 in column II) and SPOUSE (4 in column II) and CHILD (5-7 in column II). The possible ordering of records on tape is given by the P-STRUCTURE statement. Since the logical structure for this file is the same as the physical, the L-STRUCTURE statement has been omitted to conserve space.

```
*RECORD-DESCRIPTION SEGMENT NAME=HOUSEHOLD,CRD-ID=1
 VARIABLE NAME=HOUSE-NUMBER,FORMAT=1/I6
 VARIABLE NAME=STATE,FORMAT=7/A3
 VARIABLE NAME=CRD-ID,FORMAT=10/I2
          BOUND-NAME=HH,VALUE=1
          BOUND-NAME=FAM,VALUE=2
          BOUND-NAME=HD,VALUE=3
          BOUND-NAME=SP,VALUE=4
 VARIABLE NAME=COUNTY,FORMAT=12/I3
 VARIABLE NAME=LOCATION,FORMAT=15/I1
          BOUND-NAME=SW-REGION,VALUE=1
          BOUND-NAME=NW-REGION,VALUE=2
*RECORD-DESCRIPTION SEGMENT NAME=FAMILY,CRD-ID=2
 VARIABLE NAME=FAM-ID,FORMAT=19/I2
 VARIABLE NAME=INCOME,FORMAT=31/F10.2
 VARIABLE NAME=TOTAL-ASSETS,FORMAT=41/F10.2
 VARIABLE NAME=SIZE,FORMAT=63/I2
```

Four RECORD-DESCRIPTION statements appear, one for each type of record to be retrieved. Only variables to be retrieved need be described. Others can be but will be ignored if not referenced in the request. The FORMAT is "Starting Column"/"Fortran format." The BOUND is the code or value of a variable. The first four variables specified appear on every record. VALUES may be referenced by their name, e.g., HH, FAM, HD or SP for CRD-ID. The VARIABLES may be referenced by their name or a number that will be assigned by SIMS. Statements may continue on any number of additional cards. BOUNDs needn't be specified, e.g., INCOME.

```
*RD   HEAD   3   DETAIL=HEAD AND SPOUSE VARIABLES SAME
 V     TYPE-PERSON   21/I1
  B     ADULT      1
  B     CHILD      2
 V     INCOME   39/F10.2
 V     OCCUPATION   58/I2
  B     NOT-WORKING      1
  B     BLUE-COLLAR      2
  B     WHITE-COLLAR      3
  B     PROFESSIONAL      4
  B     MISSING     99
```

Parameter names may be omitted and statement names abbreviated, e.g., HEAD record description. A blank can be used as a delimiter. DETAIL description may be given as last parameter of RECORD-DESCRIPTION, VARIABLE or BOUND statements.

If OCCUPATION was not given, a MISSING value of 99 was assigned.

```
*RECORD-DESCRIPTION SEGMENT-NAME=SPOUSE, CRD-ID=4
*END
```

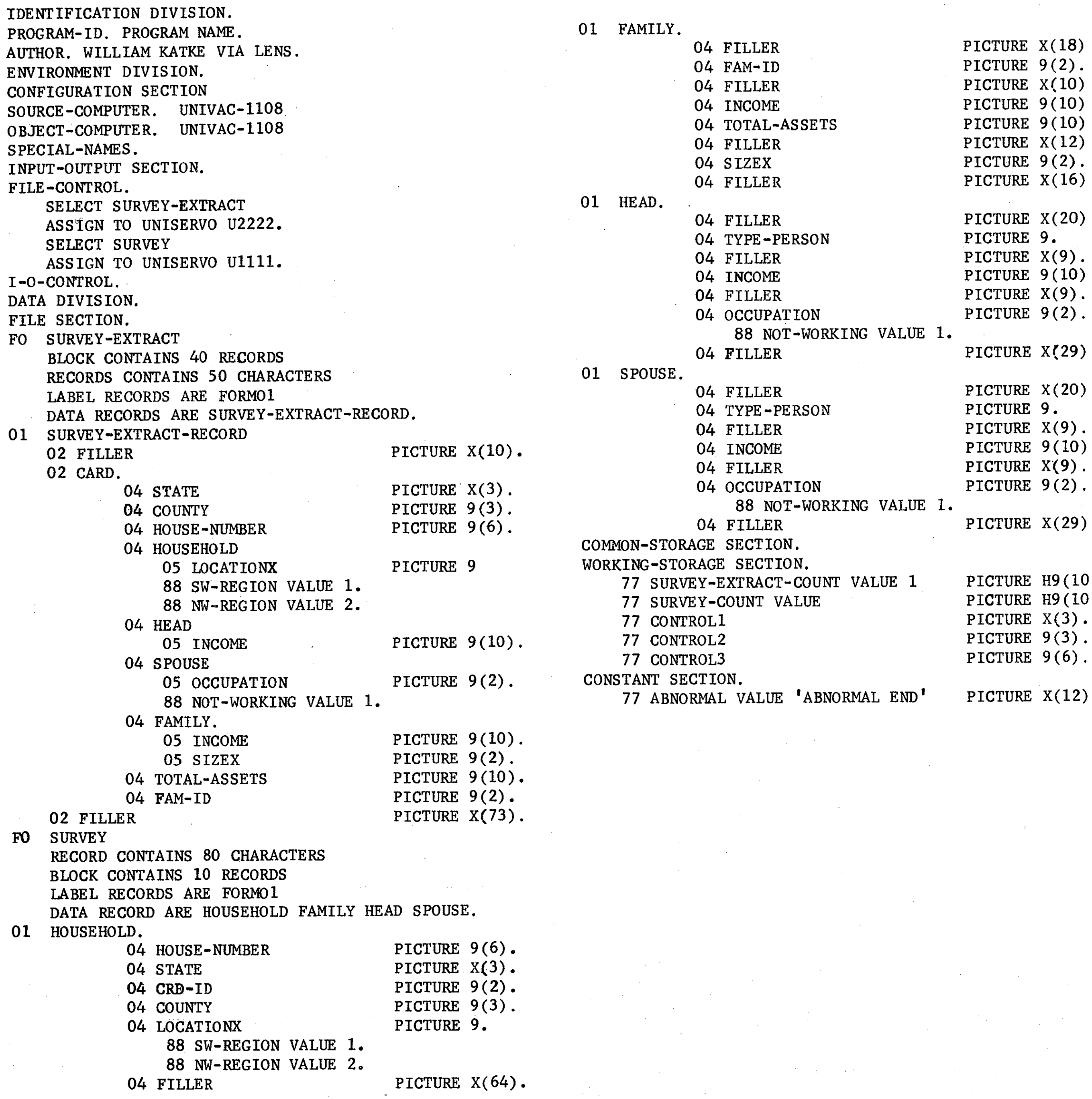
VARIABLE and BOUND statements for SPOUSE record same as head record, therefore omitted. END of SIMS request.

Figure 2—Sample SIMS request

lay either in learning a new system (rather than concentrating on his research) or hiring a programmer. Major problems have also arisen from the disparity between the programmer's neology and the social science jargon of the researcher. All this merely points to the need for a unified data management system which (1) provides a request language which can be tailored to fit the needs of a particular user class and be a comfortable and familiar means of communication with the system, (2) provides the user an interface to his routines or any other existing generalized routines which were initially exogenous to the system and (3) enables complete processing of data from outside sources, which implies provision for file creation, maintenance, generation and analysis. We have found that these three basic features are best realized by a system that provides the user with a *descriptive or non-procedural request language* and whose design reflects the *independence of data from applications programs.*

SYSTEM OVERVIEW

SIMS is presently implemented in a remote batch, time-shared environment on a Univac 1108 operating under Exec-8. Figure 1 presents a general overview of SIMS in terms of the user's input and output and the system's input and output. The descriptive requst



```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM NAME.
AUTHOR. WILLIAM KATKE VIA LENS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION
SOURCE-COMPUTER.  UNIVAC-1108
OBJECT-COMPUTER.  UNIVAC-1108
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SURVEY-EXTRACT
    ASSIGN TO UNISERVO U2222.
    SELECT SURVEY
    ASSIGN TO UNISERVO U1111.
I-O-CONTROL.
DATA DIVISION.
FILE SECTION.
FO  SURVEY-EXTRACT
    BLOCK CONTAINS 40 RECORDS
    RECORDS CONTAINS 50 CHARACTERS
    LABEL RECORDS ARE FORM01
    DATA RECORDS ARE SURVEY-EXTRACT-RECORD.
01  SURVEY-EXTRACT-RECORD
    02 FILLER                          PICTURE X(10).
    02 CARD.
            04 STATE                   PICTURE X(3).
            04 COUNTY                  PICTURE 9(3).
            04 HOUSE-NUMBER            PICTURE 9(6).
            04 HOUSEHOLD
                05 LOCATIONX           PICTURE 9
                88 SW-REGION VALUE 1.
                88 NW-REGION VALUE 2.
            04 HEAD
                05 INCOME              PICTURE 9(10).
            04 SPOUSE
                05 OCCUPATION          PICTURE 9(2).
                88 NOT-WORKING VALUE 1.
            04 FAMILY.
                05 INCOME              PICTURE 9(10).
                05 SIZEX               PICTURE 9(2).
            04 TOTAL-ASSETS            PICTURE 9(10).
            04 FAM-ID                  PICTURE 9(2).
    02 FILLER                          PICTURE X(73).
FO  SURVEY
    RECORD CONTAINS 80 CHARACTERS
    BLOCK CONTAINS 10 RECORDS
    LABEL RECORDS ARE FORM01
    DATA RECORD ARE HOUSEHOLD FAMILY HEAD SPOUSE.
01  HOUSEHOLD.
            04 HOUSE-NUMBER            PICTURE 9(6).
            04 STATE                   PICTURE X(3).
            04 CRD-ID                  PICTURE 9(2).
            04 COUNTY                  PICTURE 9(3).
            04 LOCATIONX               PICTURE 9.
                88 SW-REGION VALUE 1.
                88 NW-REGION VALUE 2.
            04 FILLER                  PICTURE X(64).
```

```
01  FAMILY.
            04 FILLER                          PICTURE X(18)
            04 FAM-ID                          PICTURE 9(2).
            04 FILLER                          PICTURE X(10)
            04 INCOME                          PICTURE 9(10)
            04 TOTAL-ASSETS                    PICTURE 9(10)
            04 FILLER                          PICTURE X(12)
            04 SIZEX                           PICTURE 9(2).
            04 FILLER                          PICTURE X(16)
01  HEAD.
            04 FILLER                          PICTURE X(20)
            04 TYPE-PERSON                     PICTURE 9.
            04 FILLER                          PICTURE X(9).
            04 INCOME                          PICTURE 9(10)
            04 FILLER                          PICTURE X(9).
            04 OCCUPATION                      PICTURE 9(2).
                88 NOT-WORKING VALUE 1.
            04 FILLER                          PICTURE X(29)
01  SPOUSE.
            04 FILLER                          PICTURE X(20)
            04 TYPE-PERSON                     PICTURE 9.
            04 FILLER                          PICTURE X(9).
            04 INCOME                          PICTURE 9(10)
            04 FILLER                          PICTURE X(9).
            04 OCCUPATION                      PICTURE 9(2).
                88 NOT-WORKING VALUE 1.
            04 FILLER                          PICTURE X(29)
COMMON-STORAGE SECTION.
WORKING-STORAGE SECTION.
    77 SURVEY-EXTRACT-COUNT VALUE 1    PICTURE H9(10
    77 SURVEY-COUNT VALUE              PICTURE H9(10
    77 CONTROL1                        PICTURE X(3).
    77 CONTROL2                        PICTURE 9(3).
    77 CONTROL3                        PICTURE 9(6).
CONSTANT SECTION.
    77 ABNORMAL VALUE 'ABNORMAL END'   PICTURE X(12)
```

Figure 3a—The precompiled MCP cobal program of the sample SIMS Request of Figure 2 (Identification and Data Division)

language consists of a Data Description Language (DDL) for describing data which is already in machine readable form to the system, and the Data Manipulation Language (DML) to specify processing to be done on the data.* The physical and logical attributes of data files described by the DDL and the processing instructions provided by the DML are made available to the DDL-DML language processor. This processor, LENS,[9] interprets these user statements, prints diagnostic messages and precompiles a job stream that is data, program and applications dependent. The descriptive request is translated into a summary consisting of

* Statements of the DDL and DML are summarized in Figures 5 and 6, respectively.

```
PROCEDURE DIVISION.
START
     OPEN OUTPUT SURVEY-EXTRACT.
     OPEN INPUT SURVEY.
                             NOTE INITIALIZATION (USER).
                             NOTE INITIALIZATION (SIMS).
     READ SURVEY AT END
         GO TO ENDPROG.
     MOVE STATE OF HOUSEHOLD OF SURVEY TO CONTROL1.
     MOVE COUNTY OF HOUSEHOLD OF SURVEY TO CONTROL2.
HOUSEHOLD.
                             NOTE SEGMENT (USER).
                             NOTE SEGMENT (SIMS).
RECORD1.
     MOVE STATE OF HOUSEHOLD OF SURVEY TO
          STATE OF SURVEY-EXTRACT.
     MOVE COUNTY OF HOUSEHOLD OF SURVEY TO
          COUNTY OF SURVEY-EXTRACT.
     MOVE HOUSE-NUMBER OF HOUSEHOLD OF SURVEY TO
          HOUSE-NUMBER OF SURVEY-EXTRACT.
     MOVE LOCATIONX OF HOUSEHOLD OF SURVEY TO
          LOCATIONX OF HOUSEHOLD OF SURVEY-EXTRACT
     PERFORM READ-SURVEY THRU SURVEY-READ.
     IF CRD-ID EQUALS 2
          GO TO FAMILY.
     GO TO STRUCTURE-ERROR.
FAMILY.
                             NOTE SEGMENT (USER).
                             NOTE SEGMENT (SIMS).
RECORD2.
     MOVE INCOME OF FAMILY OF SURVEY TO
          INCOME OF FAMILY OF SURVEY-EXTRACT.
     MOVE SIZEX OF FAMILY OF SURVEY TO
          SIZEX OF FAMILY OF SURVEY-EXTRACT.
     MOVE TOTAL-ASSETS OF FAMILY OF SURVEY TO
          TOTAL-ASSETS OF SURVEY-EXTRACT.
     MOVE FAM-ID OF FAMILY OF SURVEY TO
          FAM-ID OF SURVEY-EXTRACT.
     PERFORM READ-SURVEY THRU SURVEY-READ.
     IF CRD-ID IS GREATER THAN 2
          IF CRD-ID IS LESS THAN 7
               GO TO PERSON.
     GO TO STRUCTURE-ERROR.
PERSON.
                             NOTE SEGMENT (USER).
                             NOTE SEGMENT (SIMS).
RECORD3.
     GO TO UEP
           UEP
           HEAD
           SPOUSE
           SKIP-PERSON
           SKIP-PERSON
     DEPENDING ON CRD-ID.
HEAD.
                             NOTE SEGMENT (USER).
                             NOTE SEGMENT (SIMS).
RECORD4.
     MOVE INCOME OF HEAD OF SURVEY TO
          INCOME OF HEAD OF SURVEY-EXTRACT.
     GO TO SKIP-PERSON.
SPOUSE.
                             NOTE SEGMENT (USER).
                             NOTE SEGMENT (SIMS).
RECORD5.
     MOVE OCCUPATION OF SPOUSE OF SURVEY TO
          OCCUPATION OF SPOUSE OF SURVEY-EXTRACT.
```

```
SKIP-PERSON.
    PERFORM READ-SURVEY THRU SURVEY-READ.
    GO TO HOUSEHOLD
        FAMILY
    DEPENDING ON CRD-ID
    IF CRD-ID IS GREATER THAN 2
        IF CRD-ID IS LESS THAN 7
            GO TO PERSON.
STRUCTURE-ERROR.
    MONITOR STATE OF SURVEY
            COUNTY OF SURVEY
            HOUSE-NUMBER OF SURVEY
            CRD-ID OF SURVEY
    GO TO ENDPROG.
READ-SURVEY.
    READ SURVEY RECORD AT END
        GO TO ENDPROG.
    ADD 1 TO SURVEY-COUNT.
    IF STATE OF SURVEY IS NOT EQUAL TO
        CONTROL1
        GO TO SURVEY-OBSERVATION.
    IF COUNTY OF SURVEY IS NOT EQUAL TO
        CONTROL2
        GO TO SURVEY-OBSERVATION.
    IF HOUSE-NUMBER OF SURVEY IS NOT EQUAL TO
        CONTROL3
        GO TO SURVEY-OBSERVATION.
SURVEY-READ. EXIT.

SURVEY-OBSERVATION.
    MOVE STATE OF SURVEY TO CONTROL1.
    MOVE COUNTY OF SURVEY TO CONTROL2.
    MOVE HOUSE-NUMBER OF SURVEY TO CONTROL3.
                              NOTE INPUT (USER).
                              NOTE INPUT (SIMS).
    IF NW-REGION OF SURVEY-EXTRACT
    IF INCOME OF HEAD OF SURVEY-EXTRACT
        IS GREATER THAN 3000
    IF NOT-WORKING OF SPOUSE OF SURVEY-EXTRACT
        GO TO SELECT-SURVEY.
    GO TO HOUSEHOLD.
SELECT-SURVEY.
    WRITE SURVEY-EXTRACT-RECORD.
    ADD 1 TO SURVEY-EXTRACT-COUNT.
    GO TO HOUSEHOLD.
ENDPROG.
                              NOTE FILE-END (USER
                              NOTE FILE-END (SIMS
SYS-END.
    WRITE SURVEY-EXTRACT-RECORD.
    MONITOR SURVEY-EXTRACT-COUNT.
    CLOSE SURVEY-EXTRACT.
    MONITOR SURVEY-COUNT.
    CLOSE SURVEY.
    STOP RUN.
UEP.
    MONITOR ABNORMAL.
    STOP RUN.
```

Figure 3b—The precompiled MCP cobol program of the sample SIMS request of Figure 2 (Procedure Division)

English-like statements which indicates what and how a file or files is to be processed.

The job stream generated by LENS contains all necessary Exec-8 control cards and source statements of the precompiled program. Since the generated program and its subroutines are data and applications dependent they may be entirely in Cobol or Fortran or a mixture of both. The job stream written on drum and data descriptions of user files are made accessible to the user as an output option. This enables the user to obtain a machine readable description of his data and/or resultant program and control cards, which he may store as catalogued files, punch on cards or store on magnetic tape. This option allows the programmer user to use SIMS as a means for generating a basic program which he can then enhance to perform additional functions for which SIMS is not designed.

Figure 2 is a sample SIMS request with an explanation of each statement. Statements following and including the DESCRIPTION statement and preceding the END statement represent the data description (DDL) for the INPUT file, SURVEY. The BEGIN and END statements are SIMS control statements which identify segments of the input request. All other statements specify what processing is to be done and the output of the request; they are DML statements. Descriptions of these statements and others are provided in the following sections.

The general form of statements of the SIMS request language (DDL and DML) is:

statement-name parameter-name-1 parameter-value-1
parameter-name-2 parameter-value-2
parameter-name-3 . . .

Some statements may be complemented by one or more substatements (e.g., DESCRIPTION followed by other DDL statements) which are identified by the fact that they follow their owner. For clarity to the user, major statements have been described as having to appear preceded by an asterisk, but the LENS processor distinguishes statement types by the syntax and semantics of the DDL and DML. Parameter names may by assigned synonyms but since a parameter order is also assumed for each statement, they may be deleted if parameter values are listed in the assumed order. Since default values have been assigned to certain parameters, statements can be further abbreviated by omitting such parameters from the statement.

Figure 3 is the precompiled Cobol program that is generated by LENS from the SIMS request illustrated in Figure 2. The remainder of this paper will discuss characteristics of the DDL and DML in more detail relating to the example in Figure 2, describe in general terms the LENS system and by way of the sample Cobol program of Figure 3, describe the structure of a precompiled program of SIMS.

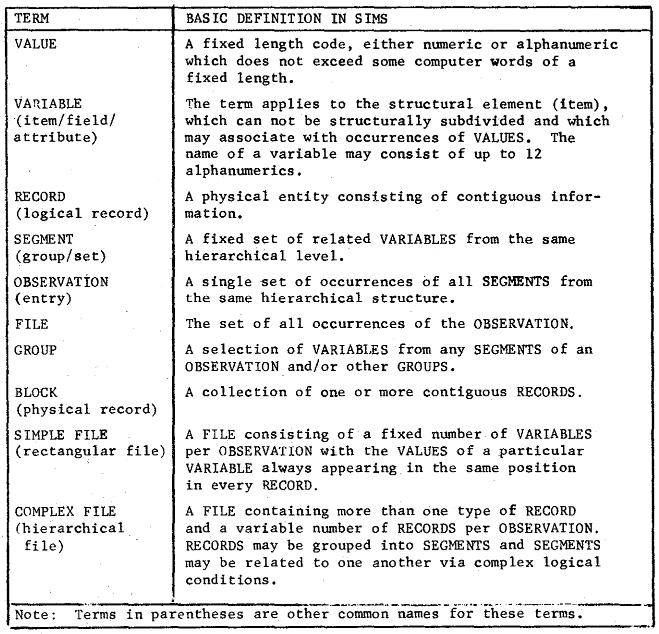
DATA DESCRIPTION-FILE CREATION

"The creation of the initial instance of a file or data base is the process of making known to the data base management system a set of entries on which it can perform other functions."[10] In SIMS the DDL is used for this purpose. Data files to be processed are assumed to be already in machine readable form, either on cards, magnetic tape, or as a disk or drum file. Card or tape files in EBCDIC or ASCII from most any computer installation can be described to the system via the SIMS-DDL. Specially formatted files, e.g., an IBM 360 tape containing floating point hexadecimal data, cannot—with the existing DDL—be described to SIMS, but through the user subroutine option could be input to the system.

A file is stored in its original physical form. If the file format is to be changed for one reason or another, a new file must be generated (see the section on *Generation*). In most instances however, SIMS can assume the physical structure remains fixed, e.g., a sequential BCD file, and have the logical structure be dependent on the particular use that is to be made of the file, e.g., random access through an indexing procedure. In this instance only one copy of the file need be made available to users, thus reducing storage.* Also, cost savings are realized since the file does not need to be reformatted to fit the particular application. Therefore, a user who is expecting to retrieve for example, 90 percent of the file, can describe the file as sequential, and if at a later date he expects to retrieve say 10 percent of the file, he can alter the data description by specifying index terms from which a table of pointers is created. The additional cost and space is only the cost of reading the file once and storing the index table.

Before discussing statements of the DDL it is necessary to define terms used in describing those statements (Figure 4). The statements of the DDL have been classified as Physical Descriptive Statements and Logical Descriptive Statements. The Physical Descriptive Statements are used to describe the storage structure or the physical structure of the data file, e.g., recording medium, blocking factor, record length and padding. The Logical Descriptive Statements describe

* Algorithms and procedures for indexing files stored sequentially have been developed and tested but have not as yet been incorporated into SIMS.[3,4]



TERM	BASIC DEFINITION IN SIMS
VALUE	A fixed length code, either numeric or alphanumeric which does not exceed some computer words of a fixed length.
VARIABLE (item/field/ attribute)	The term applies to the structural element (item), which can not be structurally subdivided and which may associate with occurrences of VALUES. The name of a variable may consist of up to 12 alphanumerics.
RECORD (logical record)	A physical entity consisting of contiguous information.
SEGMENT (group/set)	A fixed set of related VARIABLES from the same hierarchical level.
OBSERVATION (entry)	A single set of occurrences of all SEGMENTS from the same hierarchical structure.
FILE	The set of all occurrences of the OBSERVATION.
GROUP	A selection of VARIABLES from any SEGMENTS of an OBSERVATION and/or other GROUPS.
BLOCK (physical record)	A collection of one or more contiguous RECORDS.
SIMPLE FILE (rectangular file)	A FILE consisting of a fixed number of VARIABLES per OBSERVATION with the VALUES of a particular VARIABLE always appearing in the same position in every RECORD.
COMPLEX FILE (hierarchical file)	A FILE containing more than one type of RECORD and a variable number of RECORDS per OBSERVATION. RECORDS may be grouped into SEGMENTS and SEGMENTS may be related to one another via complex logical conditions.

Note: Terms in parentheses are other common names for these terms.

Figure 4—Basic definitions of data base terms

the data structure or logical structure of the data, e.g., a simple rectangular file.[11, 12, 13] Since SIMS is a "drop in" file system, i.e., accepts existing data files, the physical descriptive information is necessary.

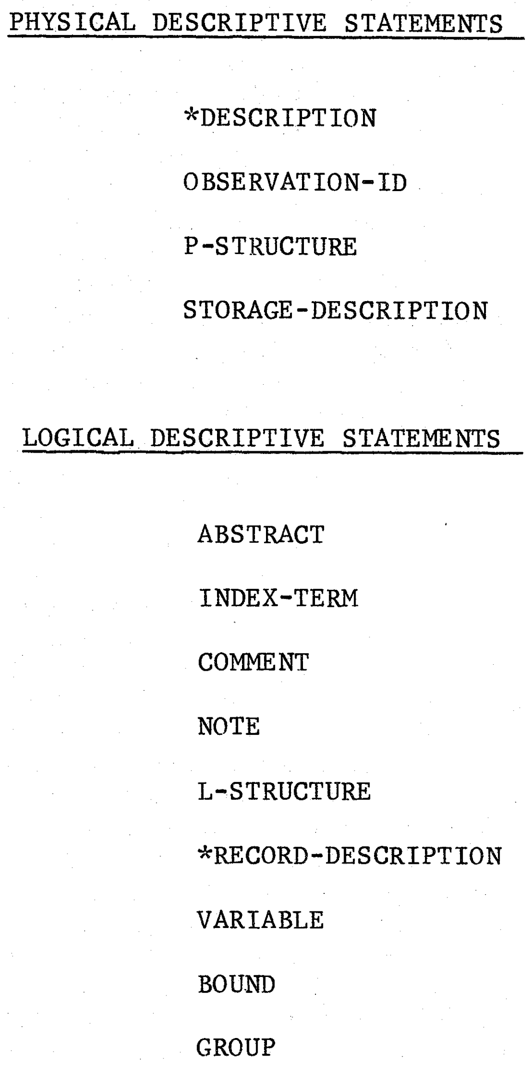
Statements of the SIMS-DDL

Figure 5 is a summary description of all statements of the SIMS-DDL. The statements given in this illustration are those that we have assigned based on experience in servicing social scientists. However, with programmer assistance the SIMS-DDL specification and vocabulary can be easily expanded to conform to other types of users. Abbreviated synonyms have also been assigned to facilitate DDL entry from a CRT or teletypewriter. The language does not address the data security problem. The Univac 1108 control language is relied upon to provide security and integrity of data files.

A DESCRIPTION statement is required for all data descriptions. The file will be catalogued by the FILE-NAME listed on this statement and this name must be specified when references to that file are made in the DML.

All records of a file must contain variables that uniquely identify an observation. Such variables must appear in the same positions in all records. The OBSERVATION-ID statement specifies this identification and provides a means for assigning a name to the observation (Figure 2).

The P-STRUCTURE and L-STRUCTURE statements are very similar. They differ only by the fact that the P-STRUCTURE statement refers to the physical structure of the file and therefore to RECORDS, whereas the L-STRUCTURE statement refers to the logical structure of the file and hence to SEGMENTS. Each of these statements describes all possible hierarchical structures occurring in any of the observations of a sequential file. The mapping of the logical structure to the physical structure is determined by the DDL processor, LENS, from information obtained from parameters of these statements and the RECORD-



PHYSICAL DESCRIPTIVE STATEMENTS

*DESCRIPTION

OBSERVATION-ID

P-STRUCTURE

STORAGE-DESCRIPTION

LOGICAL DESCRIPTIVE STATEMENTS

ABSTRACT

INDEX-TERM

COMMENT

NOTE

L-STRUCTURE

*RECORD-DESCRIPTION

VARIABLE

BOUND

GROUP

Figure 5—The SIMS-DDL statements

DESCRIPTION statements. For complex files the L-STRUCTURE statement must be present. The P-STRUCTURE statement must also be present if the physical structure cannot be derived from the L-STRUCTURE and RECORD-DESCRIPTION statements.

The physical attributes of a file are specified on the STORAGE-DESCRIPTION statement. Parameters specify storage device, recording mode, sort sequence, label information, record and block size, block padding, density, and the location and format of the record identifiers. The values of record identifiers for a given record type are listed on the RECORD-DESCRIPTION statement.

The RECORD-DESCRIPTION statement assigns a record to a SEGMENT by the SEGMENT-NAME specified, since segments are comprised of records. In Figure 2 records and segments are equivalent. In this example there are four logical segments HOUSEHOLD, FAMILY, SPOUSE, HEAD and one record for each. If there had been two records of HOUSEHOLD information then two RECORD-DESCRIPTION statements would be required, each with the SEGMENT-NAME, HOUSEHOLD.

RECORD-DESCRIPTION statements are followed by the VARIABLE statements that describe variables which are stored in that record. VARIABLE statements are followed by BOUND statements which describe values and ranges of values of that particular variable. Names may be assigned to variables and bounds.

Variables may be members of a set or group. The GROUP statement may appear anywhere within the DDL. A name may be assigned to the GROUP defined by a list containing variable and other group names. It defines a group of variables from one or more hierarchical levels and this group is defined to be at the hierarchical level of the lowest level represented by a variable. In the household example (Figure 2), if a group contained variables from all 3 levels (HOUSEHOLD, FAMILY and person (HEAD, SPOUSE)), it would be defined to be at the third or person level. Variables or items of the group from higher levels would be saved for the observation, and thus repeated for each occurrence of the lower level variable.

The INDEX-TERM statement is used to specify a list of index terms to be used for random access of a file on drum or disk. At present this statement has no effect since a random access capability has not been implemented.

The ABSTRACT, COMMENT and NOTE statements have no affect on processing of a file. They are purely descriptive and are used for documentary purposes.

DATA MANIPULATION

The SIMS Data Manipulation Language (DML) is the user's request language for specifying (1) data access to and redefinition of files previously defined by the SIMS-DDL, (2) criteria for maintenance of files that have been described to the system, (3) criteria for generating a new file from an edited file which has been described to the system and (4) analyses to be performed on files. The form of the DML is the same as the DDL, and like the DDL it is a descriptive language for stating *what* is to be done. SIMS determines *how* it is to be done.

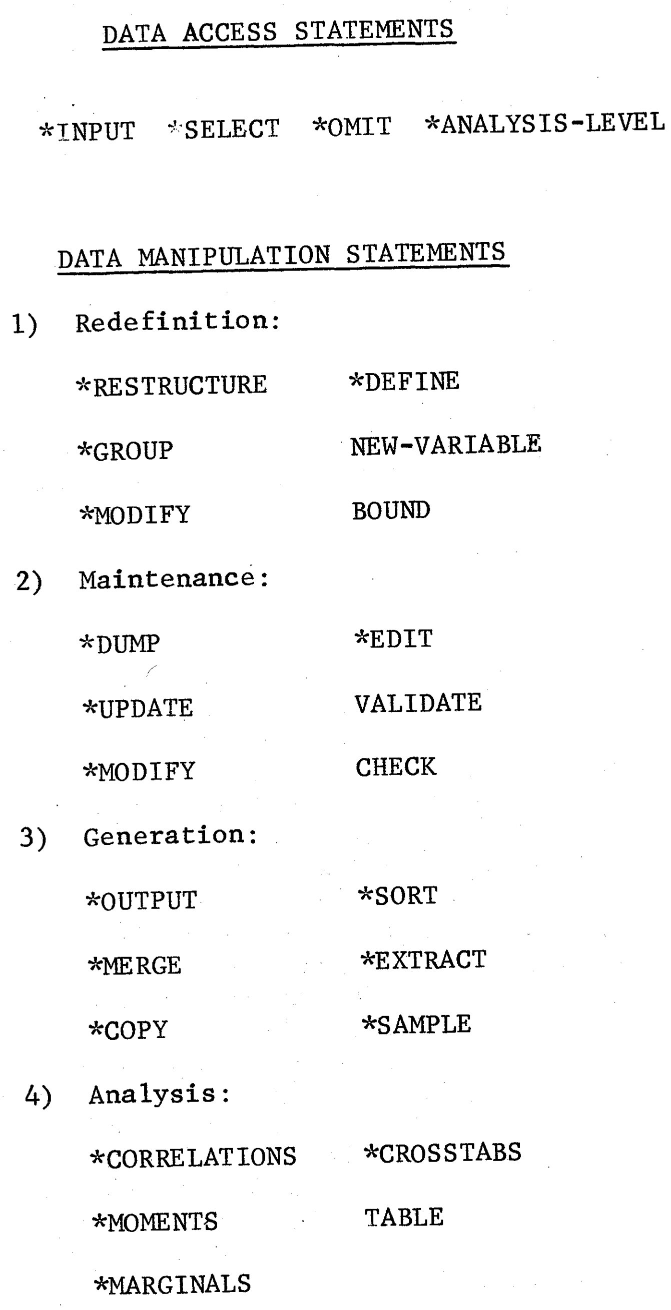
The DML is dependent on the DDL for obtaining information on both the data and storage structures, i.e., the logical and physical structures of the file. The DDL is dependent on the DML to create data descriptions for generated files (e.g., the extract file in the sample request in Figure 2), including information on newly derived items or variables. A new data description for the extracted file that is generated will be created by the system and be in a form consistent with the DDL. The two are independent in the sense that information from the DDL can be linked to a descriptive DML like the SIMS-DML or communicated directly to a programming language such as Cobol or Fortran. This independence allows the description of a file to be retained with the data. The DDL is compatible with most high level languages,[7] and the compilers for these languages could be enhanced to process the DDL.

A summary of the DML statements is given in Figure 6. Major statements are followed by their corresponding subordinate statements. With the exception of some statements which specify special analytical processing functions, statements of the SIMS-DML are of a general applications nature and could be applied to other disciplines.

Data access

The data access statements specify which file or files are to be retrieved from the data base (the INPUT statement), which entries or observations are to be retrieved (the SELECT or OMIT statements) and define what level of hierarchy shall be the basis for the retrieval (the ANALYSIS-LEVEL statement). The INPUT statement is required for all data manipulation requests and the ANALYSIS-LEVEL statement is required only if the retrieval is to be at a lower level than that of the root of the tree. If a SELECT or OMIT statement is not present, all entries or observations will be retrieved from the file.

The ANALYSIS-LEVEL statement is one means of



DATA ACCESS STATEMENTS

*INPUT *SELECT *OMIT *ANALYSIS-LEVEL

DATA MANIPULATION STATEMENTS

1) Redefinition:

*RESTRUCTURE	*DEFINE
*GROUP	NEW-VARIABLE
*MODIFY	BOUND

2) Maintenance:

*DUMP	*EDIT
*UPDATE	VALIDATE
*MODIFY	CHECK

3) Generation:

*OUTPUT	*SORT
*MERGE	*EXTRACT
*COPY	*SAMPLE

4) Analysis:

*CORRELATIONS	*CROSSTABS
*MOMENTS	TABLE
*MARGINALS	

Figure 6—A summary of statements of the SIMS-DML

redefining an observation that consists of more than one hierarchical level. If in the original file description an observation had been defined to be a household containing one or more families (each family consisting of one or more persons) and at execution time an analysis was to be based on families, then the ANALYSIS-LEVEL statement could be used to establish the family level as the observation base. Items or variables from the root level, "household," and variables from the lower level, "person" would still be associated with the appropriate family. No reorganization of the physical file takes place.

Example:

- INPUT HOUSEHOLD
- SELECT OBSERVATION IF LOCATION.IS.NW-REGION.AND. INCOME OF HEAD .GT. 3000 .AND. OCCUPATION OF SPOUSE .IS. NOT-WORKING
- ANALYSIS-LEVEL IS FAMILY

This redefinition could alternatively be accomplished with the redefinition statement RESTRUCTURE.

Redefinition

The RESTRUCTURE statement is analagous to the L-STRUCTURE statement of the DDL. Since data files are commonly used by users with varied applications and for use that is often not predetermined when the file is created, there needs to exist a means for respecifying the logical structure of the file. For the above example of a household file, if household information was not needed, the structure of the file could be respecified (deleting reference to the household information and making the family the root of the tree) via the RESTRUCTURE statement. All records containing household information would not be examined, thus increasing retrieval efficiency. The new structure specified is in effect only for the duration of the run in which the RESTRUCTURE was specified.

Another data manipulation statement for redefinition is the GROUP statement. This statement is the same as the GROUP statement of the SIMS-DDL described in the above section on *Statements of the SIMS-DDL.*

In the social sciences, transgeneration (*trans*formation and *generation*) of variables is a common occurrence in processing requests. Transgenerated variables therefore are recoded items either of an input or master file or are new variables computed as a function of input variables and other new variables. The formulation of such transgenerations is a function of input variables and other new variables. It is also a function of a particular user's methodological assumptions and, as a result, these computations are frequently of a temporary nature to be sustained only for the duration of a run. However, they may also exist permanently as an integral part of a file that is generated.

Other user-oriented systems for social science analytical applications, such as OSIRIS, STATJOB, SPSS,[14,15,16] provide specialized statements for computing transgenerated variables. For the SIMS-DML, a basic Fortran approach was taken. This was done for two reasons. First, many potential users of SIMS know or have been exposed to Fortran; and second, the user has at his disposal almost all features of the Fortran language. The standard Fortran language syntax has been appended to provide the necessary link to the SIMS-DDL and to include statementsfor performing special functions frequently used by the social scientist. The following statements exemplify the transgeneration capability and special functions of the DML.

```
IF (SEX .EQ. MALE) WAGE = HOURS*3.50
TOTAL-INCOME = INCOME OF SPOUSE +
  INCOME OF HEAD
TEMP1 = COMSUMPT + SAVINGS - LAG1
  (INCOME)
SELECT OBSERVATION IF (INCOME .BT. 0,
  3000)
```

Variables not defined by the SIMS-DDL are assumed to be local or temporary. Local variables defined must conform to the naming conventions and format of the Fortran compiler and may not be assigned names that are SIMS reserved words (such as OF, SELECT, OBSERVATION, etc.). Blanks must delimit the qualifier, OF, and may be required to delimit arithmetic operators if part of a hyphenated SIMS variable name is also the name of some other variable (e.g., TOTAL-INCOME and INCOME). Details on how SIMS extended Fortran statements and user subroutines are integrated with other statements of a SIMS request and how they are processed is given in the last section.

Associated with every non-temporary, transgenerated variable that appears on the *left* side of a SIMS Fortran arithmetic statement is a DEFINE statement. This statement defines the SEGMENT of the output file to which a transgenerated variable is to be assigned. Following the DEFINE statement are VARIABLE definitions and BOUND statements written in the DDL of SIMS. This technique provides the link between the SIMS-DML and DDL for generated or output files that contain transgenerated variables. The following DEFINE, VARIABLE and BOUND statements could apply to the transgenerated variables, WAGE and TOTAL-INCOME as defined above.

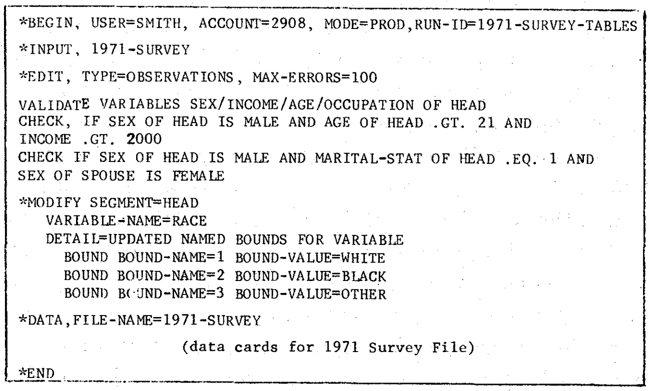
```
• DEFINE SEGMENT = PERSON
  NEW-VARIABLE NV-NAME = WAGE,
     FORMAT = F5.2 DETAIL = WAGE OF
     MALES COMPUTED AS HOURS * 3.50
• DEFINE SEGMENT = FAMILY
  NEW-VARIABLE NV-NAME = TOTAL-IN-
     COME, FORMAT = F10.2 DETAIL =
     TOTAL FAMILY INCOME OF HEAD
     AND SPOUSE
  BOUND  B-NAME=LOW        VALUE=
                             0-5000
  BOUND  B-NAME=MEDIUM     VALUE=
                             5001-
                             10000
  BOUND  B-NAME=HIGH       VALUE=
                             10000+
```

The data definition (DDL), redefinition and maintenance features must be linked closely with one another, particularly in applications dealing with data which are already in machine readable form. As the data description of a file is a function of the application for which the data are to be used, it is obviously also a function of the data itself. Errors in the file description, i.e., incorrectly specified codes or values of items or an invalid structure specification, will be detected as errors in the data during editing of the file. Therefore, the process of preparing a file description and editing the raw data could end up being an iterative process. If the source data is not available, which is often the case with survey information and other data obtained from outside sources, modification of the file description and updating of the data can only be continued to a certain point before assumptions need be made or information lost. Therefore, a data librarian or owner of a file can only go so far in describing and maintaining such files. The rest must be left to the discretion of the users.

The MODIFY statement enables a user to modify a catalogued source data description of a file for his own particular use and thereby allows him to make his own assumptions about invalid codes or observation structures. It is also used to edit the data description of a file as maintenance is being performed on it. This statement essentially dictates a link to a text editor which reads transaction statements for updating the data description. This feature is equivalent to text editing capabilities found on most other computing systems.

Maintenance

The maintenance statements of the DML provide either the data librarian or user with various means of examining the data, comparing the data with its description and correcting or updating a data file. The present version of SIMS provides only for batch maintenance of a file. The DUMP statement performs the standard utility of printing various portions of a file as specified via the parameters of this statement. The EDIT statement specifies the criteria to be used when comparing the file's description to the actual data. Editing continues until a predetermined error limit is reached (specified as a parameter of the EDIT statement) or until normal termination of the run. Options include editing of the physical structure (item or variable formats, record lengths, etc.) and the logical structure (variable values or codes, consistency checks among variables, and observation structure). Numerous output options exist and the particular option or options selected are dependent on the user's file, edit criteria and the number of errors he expects. The EDIT output is used to prepare the input to the UPDATE function. The present version of UPDATE provides only for deleting, adding or correcting entire records or entries (observations). When an interactive capability exists, specific variables will also be able to be updated without re-entry of an entire record or observation. The descriptive SIMS-DML becomes a mixture of parametric and Boolean algebraic statements in order to provide the file maintenance features. It incorporates the Fortran like IF statement using the relational and logical operators of Fortran to implement the checking necessary for updating and validation. This adaption may appear to relegate SIMS to a LOW Data Sublanguage (Boolean selection procedures) or an INTERMEDIATE Data Sublanguage (algebra oriented) as defined by Codd.[17] But the power of the SIMS-DDL is transmitted to the DML by their dependence. Hence, file maintenance can be done on variables and their logical structure in the data file. This provides the user with a tool not directly available in Fortran or Cobol, unless he is a sophisticated programmer. Figure 7 is an example of a SIMS file maintenance request.



```
*BEGIN, USER=SMITH, ACCOUNT=2908, MODE=PROD,RUN-ID=1971-SURVEY-TABLES
*INPUT, 1971-SURVEY
*EDIT, TYPE=OBSERVATIONS, MAX-ERRORS=100
VALIDATE VARIABLES SEX/INCOME/AGE/OCCUPATION OF HEAD
CHECK, IF SEX OF HEAD IS MALE AND AGE OF HEAD .GT. 21 AND
INCOME .GT. 2000
CHECK IF SEX OF HEAD IS MALE AND MARITAL-STAT OF HEAD .EQ. 1 AND
SEX OF SPOUSE IS FEMALE
*MODIFY SEGMENT=HEAD
   VARIABLE-NAME=RACE
   DETAIL=UPDATED NAMED BOUNDS FOR VARIABLE
     BOUND BOUND-NAME=1 BOUND-VALUE=WHITE
     BOUND BOUND-NAME=2 BOUND-VALUE=BLACK
     BOUND BOUND-NAME=3 BOUND-VALUE=OTHER
*DATA,FILE-NAME=1971-SURVEY
                    (data cards for 1971 Survey File)
*END
```

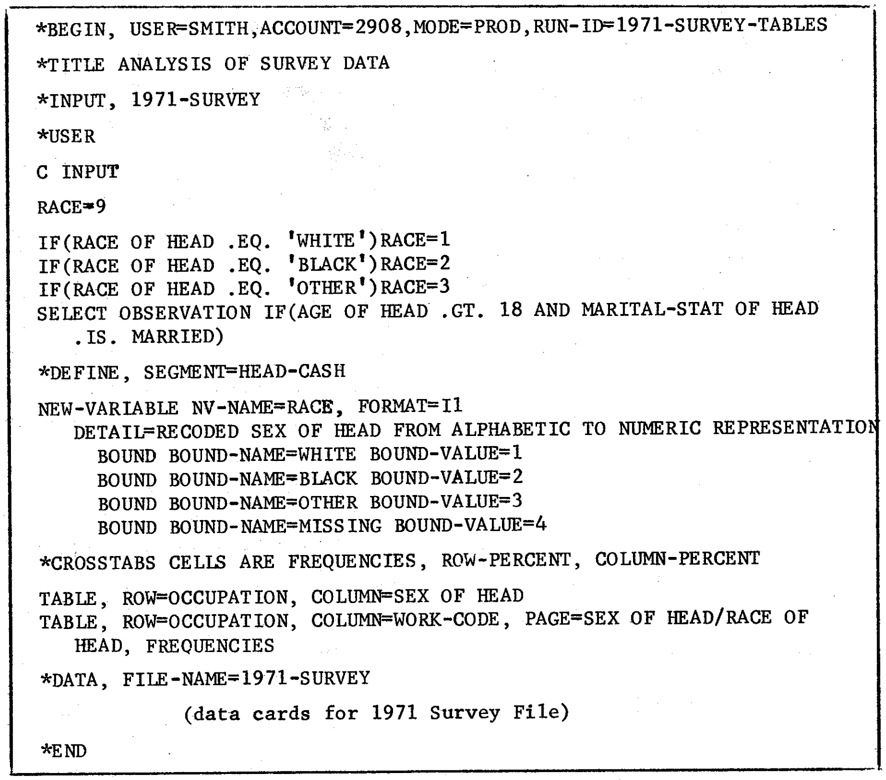
Figure 7—A SIMS file maintenance request

Generation

The generation class of data manipulation statements specifies creation of a data file or files from data already in machine readable form. If the storage structure or physical format of the generated file is to be different from the assumed storage structure of SIMS, an OUTPUT statement must be associated with every generation statement that is specified in a request. One valuable use of this feature is for defining the storage structure to be consistent with some foreign computer, thus enabling transportability of the generated file. The SORT and MERGE statements specify the parameters necessary for performing the standard utility functions of sorting a file and merging two files. The EXTRACT statement describes a sub-population of an input file that is to be generated. By using a number of EXTRACT statements and associating them with the same input file, numerous subpopulations can be created on a single pass of the original file. The COPY statement is the general case of the EXTRACT statement, since entire files may be extracted or copied. The SAMPLE statement describes a random sample that is to be taken of the input file. The sample SIMS request in Figure 2 is an example of a request for file generation.

Analysis

Since the main intent of SIMS was to provide the social scientist with a means of describing and retrieving information from complex files and not to reinvent statistical packages,[18] the present analytical capability of the SIMS-DML is quite limited. STATJOB[15] and other statistical systems and programs have been successful in fulfilling this void in SIMS. There are some analytical routines, however, which are useful aids to the user who is editing and summarizing his data. The functions specified by the CORRELATIONS and MOMENTS statements specify which variables and observations are to be included in computation of the correlation and moment or cross-product matrices for input to other analysis packages. These matrices may also be generated as data files. The MARGINALS statement describes which variables and observations are to be used in tabulation of marginal or one-dimensional frequency distributions, and similarly, the CROSSTABS and TABLE statements describe multidimensional tables of frequencies, sums, means or standard deviations of variables.[1] As with these analytical statements and others that are specified, the SIMS-DML processor analyzes what the user has



```
*BEGIN, USER=SMITH,ACCOUNT=2908,MODE=PROD,RUN-ID=1971-SURVEY-TABLES
*TITLE ANALYSIS OF SURVEY DATA
*INPUT, 1971-SURVEY
*USER
C INPUT
RACE=9
IF(RACE OF HEAD .EQ. 'WHITE')RACE=1
IF(RACE OF HEAD .EQ. 'BLACK')RACE=2
IF(RACE OF HEAD .EQ. 'OTHER')RACE=3
SELECT OBSERVATION IF(AGE OF HEAD .GT. 18 AND MARITAL-STAT OF HEAD
   .IS. MARRIED)
*DEFINE, SEGMENT=HEAD-CASH
NEW-VARIABLE NV-NAME=RACE, FORMAT=I1
   DETAIL=RECODED SEX OF HEAD FROM ALPHABETIC TO NUMERIC REPRESENTATION
     BOUND BOUND-NAME=WHITE BOUND-VALUE=1
     BOUND BOUND-NAME=BLACK BOUND-VALUE=2
     BOUND BOUND-NAME=OTHER BOUND-VALUE=3
     BOUND BOUND-NAME=MISSING BOUND-VALUE=4
*CROSSTABS CELLS ARE FREQUENCIES, ROW-PERCENT, COLUMN-PERCENT
TABLE, ROW=OCCUPATION, COLUMN=SEX OF HEAD
TABLE, ROW=OCCUPATION, COLUMN=WORK-CODE, PAGE=SEX OF HEAD/RACE OF
   HEAD, FREQUENCIES
*DATA, FILE-NAME=1971-SURVEY
            (data cards for 1971 Survey File)
*END
```

Figure 8—A SIMS analysis request

described in order to determine the validity and completeness of the request. A sample SIMS CROSSTABS analysis request with transformations is Figure 8.

IMPLEMENTATION

As stated in the introduction to this paper, a basic design philosophy in SIMS is that *data* should be separated from *applications programs*. For most applications this can be done without creating extreme inefficiencies in processing, particularly if the application programs do not operate on data at the bit level and do not assume a file structure completely foreign to the standards of the system.

SIMS uses a precompilation technique to facilitate data independence from applications programs. Both the DML and DDL are input to a precompiler which generates a job stream consisting of Cobol and/or Fortran code and the Univac operating system (Exec-8) control cards. The job stream created is a function of the user's specified request and predefined relations among SIMS statements and associated code to be generated. The association nets or the nets which describe the relation of statements and their parameters of the *existing* DDL and DML can be modified "as data" if ever the syntax or semantics of these statements need to be altered. The system needs only to be modified if *new* statements, and consequently new processing functions, are to be added to the SIMS-DML. The new statements and precompilation output that is to be generated must be described. This procedure requires an experienced programmer. Alteration of existing syntactic and semantic nets or the description of an existing SIMS-DML however, can be performed by any user. The following is a discussion of the design of the programs and subroutines that are generated by LENS and the restrictions placed upon user supplied code. This basic design is independent of the precompilation or any other implementation method.

The source Cobol or Fortran code generated from a DML statement is derived from information obtained from other DML and DDL statements and may contain calls to existing relocatable routines. An option exists to have the generated source code punched, so that it may be modified for a different computer system. A precompiled source deck could also be altered so as to achieve better efficiency in the compiled object code. This does not imply source code generated by SIMS is inefficient. On the contrary, the code generated in most instances is more efficient than what the user might write. This option would allow him to modify the code taking into account some preknowledge he may have of object code generated by the Cobol or Fortran compiler, and provides one method for a programmer to have access to data at the record level or even the bit level. Another method is discussed later.

Figure 3 shows the Cobol program that is precompiled from the request for an EXTRACT file to be generated from the input file, SURVEY (Figure 2). The data description for the SURVEY file had been catalogued or stored as a catalogued file on drum in a previous run. LENS retrieves the file description and from this generates the appropriate system control cards for loading the file description, operating system control cards (Univac Exec-8 in this case) and Cobol Data Division for reading the input file. Similarly the output file's Exec-8 statements and Cobol Data Division are determined from the OUTPUT, EXTRACT and input file description statements. The observations or entries to be extracted are determined from the SELECT and ANALYSIS-LEVEL statements, and the L-STRUCTURE and P-STRUCTURE statements of the DDL and the variables or items from the EXTRACT statement. The L-STRUCTURE, P-STRUCTURE, ANALYSIS-LEVEL and data manipulation statements (EXTRACT in this case) provide the information on the file's structure and use necessary for generating the procedural input statements that perform the mapping between the complex structure of the input and the rectangular or fixed length structure assumed for an observation. This procedure is illustrated in Figure 9 and is completed when the "observation built?" check is true.

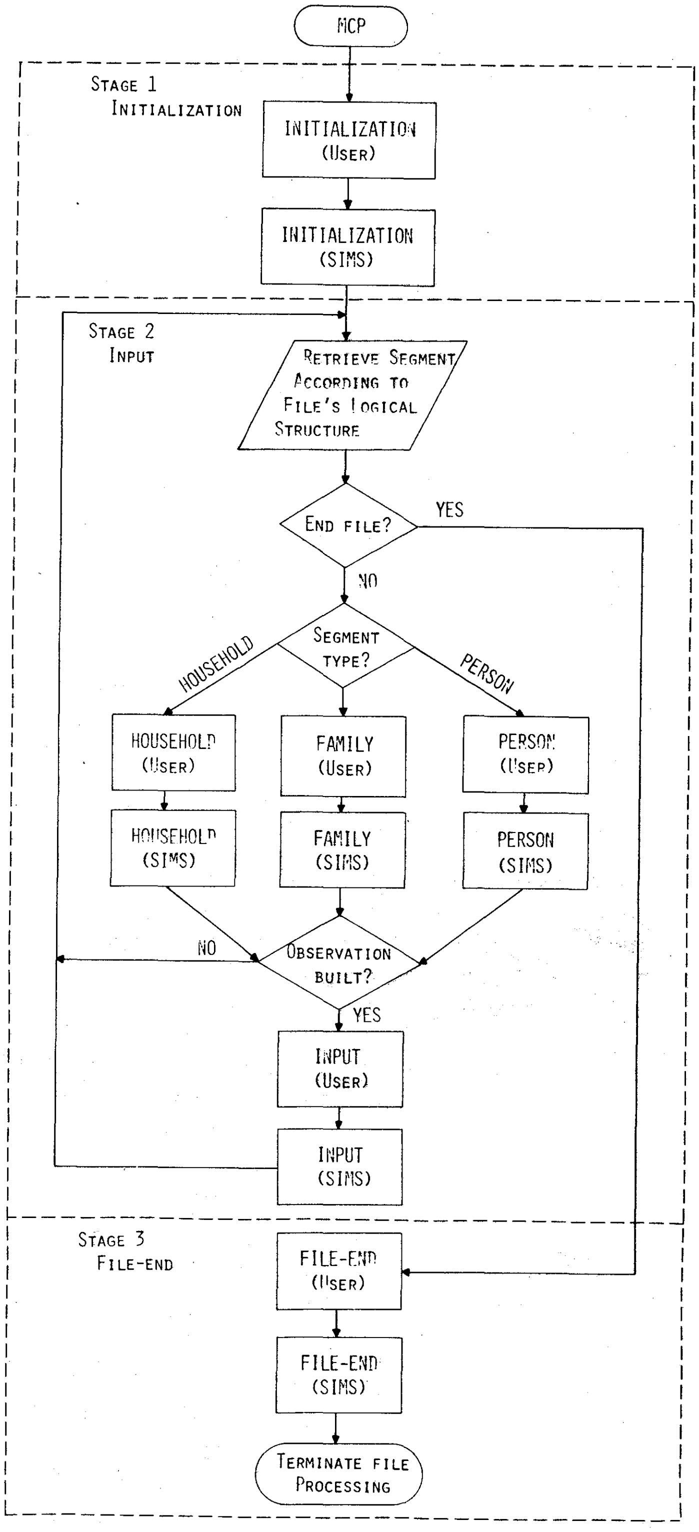


Figure 9—Logical flow of a sample MCP

The Main Control Program (MCP) (the Cobol program in Figure 3) created from the DDL and DML of the SIMS request by the LENS system is divided into three operating stages, INITIALIZATION, INPUT and FILE-END. All processing assumes only one pass of an entire file will be made in a given run. The INITIALIZATION stage of the MCP is executed before retrieving any observations or entries from the file. In this section of the MCP, calls are made to any of the housekeeping and initialization routines. The INPUT section of the MCP is executed for every entry (in the case of sequential processing) until the end of the file is reached or a predetermined number of observations or entries have been processed, in which case the FILE-END portion of the MCP takes control. FILE-END processing includes printing run summary information, etc., and terminating the SIMS run. For any form of a random access file this same 3-stage technique can be applied. If the file is indexed, then the "sequential" processing of the INPUT stage does not examine every record but retrieves only the indexed records one at a time. The FILE-END section of the MCP routine, in the random access case, assumes control when the list of indexes is exhausted. The

INITIALIZATION stage retrieves the file and constructs the index information used in INPUT processing.

The generated code for each of the three sections of the MCP contains calls to relocatable elements which may be generalized routines of the SIMS system or user supplied SIMS Fortran code (such as transgeneration statements like those illustrated in the section on *Redefinition*). For example, to produce CROSSTABS, calls are made to a crosstabulation subroutine (XTAB)[1] that tabulates observations during the INPUT stage and prints the tabulations at FILE-END. User supplied code for creating new variables for tabulation would be compiled as one of the routines of the generated run stream and the MCP of the same run stream would contain a call to that routine in its INPUT section. A simple EXTRACT, as shown in Figure 2, has no subroutine calls and consists of straight-line Cobol code. The difference between these two examples occurs because the EXTRACT request has no user code supplied and there did not happen to exist a generalized extract routine when that function was implemented.

The MCP performs all input operations on files described via the SIMS-DDL. A user has the capability of reading data via his Fortran program and transmitting it to the DML routines. The MCP builds SEGMENTS and OBSERVATIONS[7] and makes these available to a user routine (if present) before transmitting the same to SIMS routines for processing. In sequential retrieval from hierarchical files, this construction enables the user to examine information at any hierarchical level and perform operations on the data which may not be presently available through the SIMS-DML. The same is true of computations or complex logical checks that he may wish to perform after an observation is constructed. For an indexed file only a routine related to an observation is possible, since the hierarchy "disappears." Figure 9 illustrates the main construction of a model MCP by showing a possible flow of a generated program to process the SURVEY file mentioned earlier. User supplied routines at all three stages of processing and for each type of segment are illustrated. The retrieval of the various segments of an observation is a function of the L-STRUCTURE specified in the DDL. The specific L-STRUCTURE of the SURVEY file is immaterial to this discussion, since each of the segment routines, HOUSEHOLD, FAMILY and PERSON, are called whenever the associated segment is detected.

The user can input data not described via the SIMS-DDL by modifying a punched deck of the MCP, as discussed above, or by including input statements in any of the user supplied subroutines. A complete knowledge of the processing flow of the generated MCP and communication among the MCP and its subroutines is required. The special data may appear on cards or tape. The DATA statement, identified by a segment name (e.g., HOUSEHOLD, FAMILY, PERSON) or the user subroutine names (INITIALIZATION, INPUT or FILE-END), must precede the card data that is to be read by the subroutine listed. The programmer-user must adhere to the restricted logical input unit assignments, layout of COMMON and subroutine calls or else erroneous results may occur or the run will terminate abnormally. Special output, resulting from user output statements in a user subroutine, is produced using an analogous procedure as described above for special input.

SUMMARY

Design and implementation features of a data base management system for social science applications have been presented. A particular class of user was examined, his needs brought to focus and the requirements for SIMS established. These requirements were continually reviewed during development of the system and eventually evolved into the system features described in this paper.

The basic system and language design provided the flexibility necessary for implementing new features as they became known. It was possible to easily incorporate generalized analysis subroutines developed independent of SIMS, make use of file handling routines of the host operating system when needed and add or alter statements of the DDL or DML. With a minimal knowledge of the system, the user benefits from these design features by being able to incorporate his own routines and alter the request language. Potentially he can create his own personalized data base management system.

A user is provided a complete file handling facility for file creation, maintenance and generation. Additional familiarity is provided the user by the fact that he can process data he has created previously for other purposes without reformatting the data in some new system format. This facility answers in part Gosden's complaint, "Such systems [for receiving foreign files] have not been reported in the general literature and the common data management systems do not appear to have given more than minimal attention of this problem. This is probably because they have made their success in application areas operating on well-organized data bases consisting of files especially constructed for a set of applications where they do not need to provide facilities for files . . .[19] Since file processing functions bridge all application areas, the file creation, maintenance and generation functions in SIMS apply to other disciplines as well.

SIMS via LENS precompiles a descriptive user request and generates a job stream that is data, program and application dependent. Despite the complexity of the generated code, the data description and manipulation languages are not complex and are not order dependent. Therefore the user can supply his request in what is a logical order for him, not the assumed order of the system. The syntax, semantics and completeness can be checked when the *entire* request has been read and while the *entire* request is being processed. The program that is generated is efficient because it is custom tailored to the request.

LENS, the DDL and DML processor for SIMS, is a language processing system written in Fortran and Cobol. In converting SIMS to another hardware configuration only the Fortran and Cobol routines need be compiled and debugged to implement LENS and consequently SIMS. Some of the generated code would need to be modified, i.e., Univac 1108 Exec-8 statements. This is done by changing generation macros within LENS. Thus with the LENS implementation, transportability and modularity of SIMS is enhanced.

ACKNOWLEDGMENTS

The SIMS system has been funded by the National Science Foundation (grants GS-1992 and GS-3307) and the University of Wisconsin. This project has been under the faculty direction of Professor Dennis Aigner with Max Ellis directing the system design and implementation. Significant contributions in development of the system have been made by William Katke, James Olson, Shou-Chuan Yang and Kenneth Nelson. These persons with Max Ellis have designed the system, its user interface and programs.

The critical advice of Professors Dennis Aigner, Tad B. Pinkerton, Larry E. Travis and Richard L. Venezky in preparation of this paper is greatly appreciated. Special thanks are due to our highly-spirited secretary Bonnie Balcer.

REFERENCES

1 DATA AND COMPUTATION CENTER
XTAB-XTABRUN: A cross-tabulation package (user-programmer's manual)
Data and Computation Center University of Wisconsin 1969

2 M E ELLIS K H NELSON
A data description language for hierarchical data files
1970 ACM-SIGFIDET Workshop on Data Description & Access
ACM 1970 pp 87-106

3 S C YANG
A search algorithm and data structure for an efficient information system
1969 International Conference on Computational Linguistics (Stockholm) preprint no 51

4 S C YANG
HAICS: A data structure for efficient search and retrieval
Data and Computation Center University of Wisconsin 1971

5 S C YANG Y PAL
A design of a pupil data base subsystem for decision-making in public schools
Data and Computation Center University of Wisconsin 1972

6 E H SIBLEY G C EVEREST
Critique of the GUIDE-SHARE DBMS requirements
1971 ACM-SIGFIDET Workshop on Data Description, Access and Control ACM 1971 p 103

7 S C YANG M E ELLIS
SIMS-DDL: A data description language for social science applications
Data and Computation Center University of Wisconsin 1972

8 M E ELLIS
Fortran coding standards
Data and Computation Center technical paper (TP-15) University of Wisconsin 1970

9 W F KATKE
LENS: A language implementation system
Data and Computation Center University of Wisconsin 1972

10 CODASYL SYSTEMS COMMITTEE
Feature analysis of generalized data base management systems
ACM 1971 p 377

11 CODASYL
CODASYL data base task group April 1971 report
ACM 1971

12 C J DATA P HOPEWELL
Storage structure and physical data independence
1971 ACM-SIGFIDET Workshop on Data Description, Access and Control ACM 1971 pp 139-168

13 R W ENGLES
An analysis of the April 1971 data base task group report
1971 ACM-SIGFIDET Workshop on Data Description, Access and Control
ACM 1971 pp 69-91

14 ICPR AND ISR
OSIRIS II user's manual
University of Michigan 1971

15 MACC
STATJOB reference manual for the 1108
Madison Academic Computing Center University of Wisconsin 1970

16 N H NIE D H BENT C H HULL
SPSS: Statistical package for the social sciences
McGraw-Hill 1970

17 E F CODD
A data base sublanguage founded on the relational calculus
1971 ACM-SIGFIDET Workshop on Data Description, Access and Control ACM 1971

18 J R OLSON
SIMS: A social science information system
Data and Computation Center University of Wisconsin 1972

19 J A GOSDEN
The conceptual requirements for a management information data bank
IFIP Congress 71 Booklet TA-5 p 55

A data dictionary/directory system within the context of an integrated corporate data base

by B. K. PLAGMAN

Federal Reserve Bank of New York
New York, New York

and

G. P. ALTSHULER*

Chase Manhattan Bank, N.A.
New York, New York

INTRODUCTION

The evolution of information

The evolution of commercial business applications has gone from simple repetitive accounting to extremely complex information systems, and has been characterized by decentralized control of data. Most corporations began their automation efforts by computerizing the repetitive operations of the clerk and bookkeeper. These may be referred to as Accounting or Operational Level Systems. With the advent of second and third generation hardware and software, it became feasible to collect relevant Accounting Level Systems under a functional umbrella and to relate these applications in some manner or means with the objective of providing management with a slightly broader scope of information about the business operation. Systems supporting this level of information can be termed Functional Level Systems. In the banking environment this has manifested itself in information systems (sometimes labeled MIS) dedicated, for example, to Deposits or Loans. These systems contain within them subsystems which are in essence Accounting Level Systems, such as Demand Deposit Accounting or Installment Loans.

In recent years, another level of informational need has arisen. Corporate Level Systems are designed to provide management with information that will impact decision-making on a corporate-wide basis. The various Functional Level Systems are required to supply income, expense, cost and profit-related data to the Corporate Level Systems to facilitate the reporting of this higher level of information. (See Figure 1.)

It has been this hierarchy of informational needs, the Accounting, Functional and Corporate Levels, that systems have evolved to support. (The term evolution is emphasized because of the decentralized nature of development.) It is through no fault of the system designers, however, that when the need for Corporate Level Information had arisen, the hierarchy failed. Indeed, some have experienced problems at the Functional Level before attempting to develop Corporate Level information. The problems of the implementation of these systems under a decentralized approach are characterized by the following difficulties:

Redundant data

As each Functional Level system evolved it created and maintained its own data files. To continue the examples in the banking industry, if the Deposit system and the Loan system were dealing with the same customer, typically the customer's identification data (e.g., name, address, social security number) were stored redundantly in each system. Furthermore, if the bank were supporting a Central Customer Information File, typically the data would be stored all over again. This is but one type of redundancy: Duplication of data because it is actually needed in two places and

* As of September 1, Diebold Europe S.A., 6000 Frankfurt (Main), Feuerbachstrasse 8, West Germany.

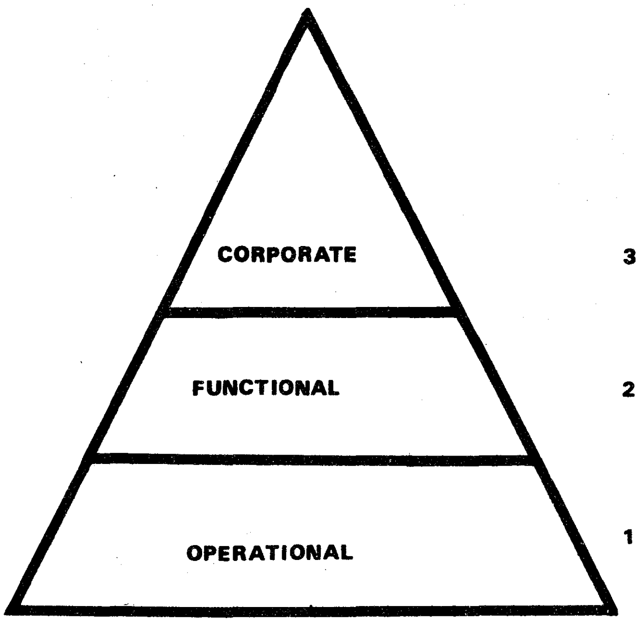


Figure 1—Levels of information

the current technological trade-offs dictate redundancy. However, the problem has been compounded in large organizations where communication among staff has failed and inadvertent redundancies have been introduced in data systems. (We shall return to these two types of redundancies.)

Inconsistency/incompatibility of data

As data files were created and maintained by each Functional Level System, large corporations found themselves in a position where each system was dealing with data which was either inconsistent or incompatible with the data of other systems. At one high level meeting of a large computer manufacturer, attendees had brought what were purported to be comparable financial reports from their respective operating divisions, only to realize that the reports were unrelatable. The definition of terms and therefore the data collection in the various systems were so diverse as to render the comparison of information useless.

Software data dependence

As each functional system was designed and implemented, utilizing its own data files, the supporting software became bound to the data which it manipulated. This in turn created high maintenance costs in instances where the dynamic aspects of data manifested themselves causing large reprogramming and recompiling efforts. Witness, for example, the situations in most large financial institutions just a few years back when the Standard Industrial Codes were introduced into data files.

The integrated corporate data base

These major problems and others not mentioned have one thing in common. They all stem from and are fostered by decentralized control over data. The concept of allowing each application to "own" and control the data which it manipulates is the underlying cause of difficulty. One potential solution to these problems is the implementation of the Integrated Corporate Data Base concept (ICDB). This basically means treating data as a corporate resource just as machines and money are. An ICDB can be formally defined as:

> The consideration of the collection, storage and dissemination of data as a logical, centrally controlled and standardized utility function.

The elements of the integrated corporate data base

It should be emphasized at this point that an ICDB is not a system, it is a concept under which systems should be implemented. There are five sub-systems or elements of such an ICDB concept:

1. The Data Bank—the logically centralized repository of all the data utilized in a corporation.
2. The Data Dictionary/Directory System—the repository of all definitive information about the Data Bank such as characteristics, relationships and authorities.
3. The Data Base Administrator—a human function responsible for coordination of all data related activities.
4. The Data Base Management System—a software function performing the storage, retrieval and maintenance of data.
5. The User/System Interface—the necessary subsystems to permit multiple classes and types of users to direct the system to effectively structure the available data into information and thus communicate with and fully utilize the resources at their disposal.

Two alternative architectures

1. Data base management system driven ICDB—In an ICDB, where the Data Dictionary/Directory System is subordinated to being solely a source of information, the Data Base Management System is the component that causes data to flow through the system.

In this approach, the Data Base Management System will have its own sources for obtaining information about the data stored in the Data Bank. Users would be required to specify meta-data, that is, data about data, in specialized formats to the Data Base Management System. This meta-data required by the DBMS is usually limited to directory type data, such as the internally necessary items that it must know to effectively store, retrieve and maintain data to, from and in the Data Bank. Dictionary type data that is of special interest to the users of data (e.g., source of data, type of data, etc.) is not collected nor stored. It remains, if at all, a function of a special collection system to collect, store, and maintain this type of data about data. It becomes a matter of redundancy, if not contradiction, in instances where the needs of the DBMS and the user overlap. Where should meta-data such as the length of a data element be stored? How should it be stored, as the user sees its length (absolute value) or as the DBMS sees its length (relative displacement)?

It is perhaps most ironic that this first architecture of an ICDB permeates within it precisely the maladies that the ICDB is designed to eliminate. Data redundancy is built in by creating two sources for collecting, storing and maintaining meta-data, albeit one is for the machine and one is for the man. Inconsistency/incompatibility cannot be insured against by allowing the control over data definitions to reside in the software that executes on the data. There is no assurance that what the user is told via a free standing DD/DS is actually what is being stored and maintained in the Data Bank. Finally and possibly most important, Data Base Management Systems which store meta-data within themselves cause the user's data to be dependent on the DBMS. This is in essence a form of data dependence. Storing data base definitions internally in a DBMS precludes the ability to change from one DBMS to another. This issue should be an important one in today's competitive market where it is so difficult to choose the "best" DBMS available. When should an installation be locked into the use of one DBMS? (See Figure 2.)

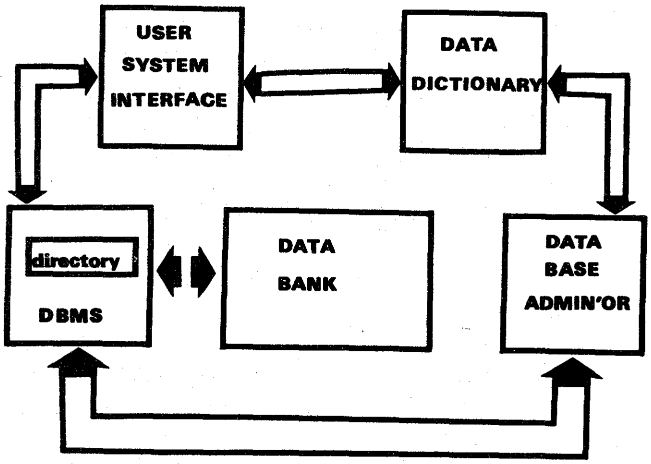


Figure 2—DBMS driven ICDB

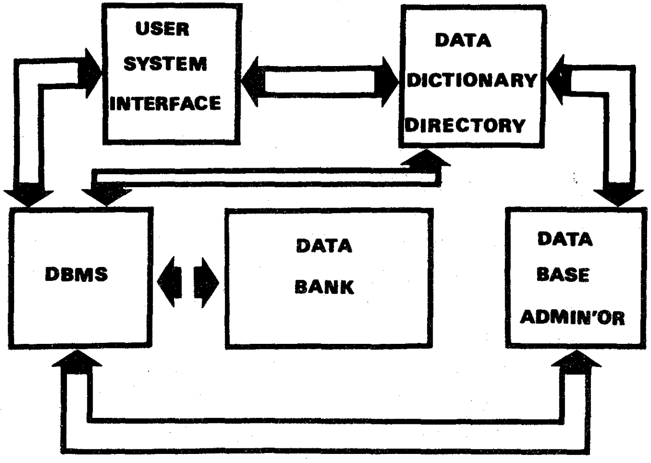


Figure 3—DD/DS driven ICDB

2. The alternative that this paper proposes and details is an ICDB that is driven by a Data Dictionary/Directory System. In this architecture, the DD/DS is the central source of control of all data specification and data flow. The DD/DS contains, in a logical sense, all the meta-data in the ICDB. The Data Base Management System requests from the DD/DS the necessary information it needs to execute physical to logical mappings made necessary by requests of users. The users of data consult the Dictionary/Directory for information about data that they might need in making queries, writing programs or simply knowing what data is available and where it is. Thus the DD/DS serves the User/Systems Interface and Data Base Management elements of the ICDB. Further, by definition the DD/DS defines and describes the Data Bank and is the most important tool of the Data Base Administrator. (See Figure 3)

DETAILS OF THE DATA DICTIONARY/ DIRECTORY SYSTEM

The objectives of design

The design of a DD/DS will generally address two objectives, but in varying degrees of emphasis.

- Collection and dissemination of data—This entails the function of supplying the users of data with meta-data, and providing the DBMS with the information it needs to operate.
- Establishment of standards—This addresses the need of establishing standards for such things as data naming, usage and coding conventions.

The amount of emphasis placed on each of these objectives will have a profound effect on specific aspects of the design. If standardization is emphasized then the scope of what is allowable in the system can be narrowed down to standardized elements. If collection and dissemination is the primary objective, then a wider range of possibilities must be provided for, since what is, must be collected as opposed to what should be. The degree of emphasis to be placed on either of these objectives will usually be indicated by the individual characteristics of the systems environment. For example, a highly decentralized systems structure would probably require emphasis on collection and dissemination, while a centralized environment with stricter controls would more aptly lean toward emphasis on standardization. Regardless of emphasis, however, both objectives must be addressed and met to the degree necessary.

Functional requirements

The following is a statement of the functional requirements for a DD/DS within the context of the ICDB concept.

1. Clear specification of data—Most data processing installations are engaged in the continuing activity of developing systems to support the development of Corporate Level Information. In order to design the appropriate information flows for future management support, the system designers must be able to analyze existing data and data flows precisely. The emphasis here is in specifying and describing data so that it would be clear to anyone interested, exactly how to utilize the data element. A DD/DS designed to maintain clear specifications of data will provide this necessary support.
2. Simple, selective retrieval—The users of the data should be able to select precisely the items of meta-data that are of interest to them and review only the specifications of those individual items. This requirement implies the ability to access meta-data on the basis of standard labels (keys) as well as by associative searching.
3. Inconsistent redundant data analysis capability—Identical or very similar items of data occurring in data files in different parts of the business indicate potential areas for concern. Detection of such cases must be possible so that each may be studied in detail. Those responsible for the data involved (the DBA in general and the user in particular) must have a tool to enable them to determine if inconsistency/redundancy does actually exist. In general there are two types of redundancies. Technical redundancy is knowingly built into a system because of existing technological trade-offs. Periodically these trade-offs should be reevaluated, and if it is decided that redundancy should be tolerated, consistency between duplicated items must be maintained. The second type of redundancy, the inadvertent type, should never be tolerated and where possible should always be eliminated when detected. Unless one has precise specifications of data and can readily compare these specifications, it is very difficult to detect cases of possible inconsistency and/or redundancy.
4. Knowledge of the location of data—When a systems analyst requiring certain data for a particular application is not in a position to know the existence of relevant data, redundancy may occur. To avoid this, the DD/DS must include not only a clear specification of the data but also a statement of where it exists and when appropriate how it can be retrieved.
5. Determination of data users—It is possible that an individual in one functional area has been an unofficial user of data generated in a second functional area. When the functional area that generated and maintained this data no longer has a need for this particular data item it will, not knowing of the impact on other users of the data, delete the data items. To avoid undesirable deletion in such circumstances, the DD/DS must include means for registering all users of each data item.
6. Assignment of responsibility for data integrity and specification—In general the Data Base Administrator carries the ultimate responsibility for insuring the correctness and complete-

ness of the data itself and the specification data. He must, however, delegate this responsibility to users of data. When, for example, data is transmitted from a remote location to a central point and modified there in one way or another, confusion may arise regarding exactly who is responsible for specifying the data that is processed centrally, and who is responsible for the correctness and completeness of the data per se. The DD/DS must clearly delineate both of these responsibilities.

7. Support of the DBMS—The majority of data management systems on the market today include directory modules, which contain necessary descriptive data which the data management software needs to execute requests. The DD/DS should contain this information and be able to supply it (via an appropriate interface) to the DBMS upon request. This will preclude storing such data twice and insure the integrity of the DD/DS, to the extent of the information transferred to the DBMS for use in executing requests.
8. Support data validation—Most corporations with highly transaction-oriented businesses (e.g., Public Utilities) require elaborate data validation procedures. The DD/DS should provide the facility to specify the validation requirements for each data item specified and be able to pass parameters to appropriate software modules (which may or may not be part of the DBMS) which then actually perform the validation, and produce error reports. By centralizing validation procedures in this manner, greater control over input can be realized as well as effecting a savings in terms of software development, since each application will no longer be required to develop its own validation routines.
9. Open-ended design—The design of the DD/DS must be such that it allows for graceful expansion to satisfy new and varied requirements of data users, and Data Base Management Systems. The new and evolving nature of data base technology dictates that any element of the ICDB that is to remain useful over any extended period of time must be designed so that it can adapt to changing needs. For example, should the need arise to support a second DBMS there should be no problem of compatibility. The open-endedness of design will have insured against any such problem. (The next section will indicate a design that can satisfy these requirements.) Another example along different lines might be when the need should arise to include a new type of specification for data (e.g., including data element names for a new programming language), the DD/DS should be able to accommodate the specifications for it.

Data management system independence

Many corporations are currently faced with the dilemma of choosing from among available data management systems to satisfy functional requirements of specific applications, while at the same time realizing that they do not have answers to these pressing questions.

- Can the data management system chosen be used by all other applications that will need such a system?
- Will the data management system chosen be able to support the ICDB concept?
- Will the next data management system chosen be compatible with the one currently being considered?

The most perplexing aspect of these questions is that there are no clear-cut answers. Nevertheless, one possible solution may be by incorporating within the ICDB design a software interface between the DD/DS and the DBMS. The interface would be adaptable to the requirements and formats of any DBMS. In essence what any DBMS requires in terms of data specification is a data control block containing relative displacement, access method and mode of representation, among other things, in certain pre-defined formats. The interface would provide these control blocks from the data available in the DD/DS. Thus, one need not worry about converting from one DBMS to another in terms of converting data, but rather in terms of adapting the interface. It should be noted also, that to effect complete program independence from the DBMS, it would be necessary to incorporate into the interface a standard program interface by developing a language to be used in making requests from the Data Bank. This second aspect is much more difficult and complex than the first, since a DBMS also comes with its own language to be used in making requests. Thus the interface would also need the capability to map from the standard language request to the DBMS language request.

The contents of the dictionary/directory

The characteristics or attributes of data items stored in the Data Bank make up the contents of the dictionary/directory. In order to allow for maximum flexibility and minimum redundancy in organizing the DD/DS its contents may be divided into two parts, variable and non-variable.

- Non-variable meta-data—This refers to attributes which cannot change from one use of the data element to another. A data element's name and description, for example, would usually be non-variable.
- Variable meta-data—This refers to attributes which can change for one use of the data element to another. The representation (binary string as opposed to character string) of a data element would be classified as variable.

By utilizing this dichotomy of meta-data, the DD/DS provides data specifiers with the ability to define similar data elements by specifying the attributes that are similar only once and then defining the variable characteristics. This referencing is then extended within the DD/DS and minimizes redundancy by storing non-variable attributes only once.

Attributes to be collected

The exact list of attributes that will be collected and maintained in the DD/DS largely depends on the relative emphasis placed on either of the two objectives, collection/dissemination and standardization. Regardless of which approach is taken, however, the following classifications of attributes of data must be addressed at one level of detail or another.

1. Identification of data—In this category it is necessary to record the varying nomenclature used to refer to a data element. A DD/DS should provide for specifying of an official name, program language names, and synonym names. In addition this section should include a textual description of the data element to be understood by any user of the system. It is interesting to find that most DD/DS also provide for a rigorously structured technique to be used in describing data elements. This description or some part of it is then used as the official name. One such notable technique is a language developed at IBM, sometimes referred to as the "OF Language". Specifications relating to the security of data would also fall within the general category of identification. It should be noted that there are two types of security: content security refers to security for access to the values of a data element, and descriptive security provides for security for accessing the meta-data relating to a data element.
2. Source of data—This category of meta-data relates to the source of values for elements of data. Data element values can be stored as received or generated from other source data. If values are obtained from source documents these should be enumerated and identified. Those responsible for the entries in source documents should also be included to insure integrity of data values at their source. In cases of generated data, the algorithms or formulae used to derive the data item in question should be specified.
3. Type of data—There are basically two types of data elements: those that measure or represent amounts, and can be termed quantitative, and those that tell something or contain qualitative meaning and can be termed indicative. Depending on the need for precise detail, this category can be broken down into as many as a dozen different sub-types within these two types. Some examples are: codes, names, amounts and dates.
4. Use of data—There is a wide variety of both essential and non-essential attributes that fall within this category. Some of these are where-used specifications, both internal and external, and length and representation data. Other attributes that might be included are the status of the data element (e.g., proposed or approved) and the names of programs which physically utilize the data element.
5. Qualification of data—This category indicates the exact circumstances or pretext under which the data element is intended for general use. The accuracy of quantitative elements would be indicated and time dependencies such as frequency of update might be included. Of considerable importance in the DD/DS would be the accessing key of the record (i.e., the logical unit of retrieval) in which this data element appears. (The key may be the data element itself, in which case either it is actually the accessing key or the file is inverted.) Edit masks would also appear in this category, as well as data validation requirements.
6. Relationships of data—In this category a specifier of data would enumerate what relationships this data element had with others. Membership in groups, arrays, sets, networks and any other type of data structure would be specified. In addition special relationships such as precedence and coincidence (e.g., "condi-

tion: if data element A is equal to value X then data element B must also be value X, or if in order for data element B to contain a value other than null, data element A must not be null"), could be identified.

IMPLEMENTING A DD/DS IN THE CURRENT ENVIRONMENT

The foregoing is a detailed statement of requirements for a DD/DS that is designed to support the ICDB concept. Two critical issues remain unsolved, however. How would an installation go about installing such a DD/DS and how would this lead in the direction of the ICDB concept?

DD/DS as step one toward an ICDB

The DD/DS in conjunction with the DBA is the logical first step for a corporation to take in beginning to implement the ICDB concept. This concept is manifested in treating data as a corporate resource to be shared by all users. The clear specification of data, ready access to such specifications and central administration of data using these tools are the necessary prerequisites to sharing data between functional areas. The ability to share data is clearly the first step toward integrating data across functional lines.

DD/DS may be most justifiable in its own right

A more pragmatic reason for beginning to implement the ICDB concept with the DD/DS is that in many cases a DD/DS can prove to be cost justifiable because of the immediate benefits that can be realized from it. The degree to which these benefits are attainable in the short range depend on the particular environment involved. Thus, in general terms, a typical large EDP installation stands to benefit from a DD/DS in the short range in the following areas.

1. Redundancy—The initial DD/DS will at least uncover the instances of redundancy. Those that are inadvertent in most cases can be eliminated immediately, while those that are technological will usually be, at least temporarily, tolerated.
2. Data transfer—The initial DD/DS will make the process of transferring data from one functional area to another easier to accomplish. By simply having clear data specification available to both parties involved in the transfer of data, the ability to effect these transactions will be enhanced.
3. Locating relevant data elements—The initial DD/DS should provide systems analysts with a formidable tool in locating those existing data elements which have relevancy to systems which they are designing. This facility will address the problem of inadvertent redundancy at its source.
4. Documentation—Depending on the level of sophistication, the initial DD/DS should provide some documentation support to programmers and/or analysts. One relatively simple design installed in a manufacturing environment provided for automatic generation of record layouts.

The data base administrator and the DD/DS

The DD/DS is primarily a tool for the DBA in carrying out his duties and responsibilities. A full and detailed description of the DBA and his function is not proper in this place. What should be clear at this point, however, is that as the DD/DS is being installed, preparations should be made to establish the DBA in his proper place. The initial task of the DBA will largely be related to the administration of the DD/DS. As the DD/DS evolves and grows into supporting the full ICDB concept, the DBA and his functions will evolve and expand in a complementary fashion.

Evolutionary development of the DD/DS

It cannot be over-emphasized that the initial DD/DS is not the end but only a means to an end. The initial DD/DS must have the inherent capability to evolve into a fully grown DD/DS supporting an ICDB concept. The open-endedness requirement stressed at the outset of this paper should insure that this capability exists. There are at least two specific areas where this aspect of the design can be taken into account.

1. File design within the DD/DS—the DD/DS should contain files which are organized around the concept of the data element. The ICDB concept emphasizes the sharing of data. This is accomplished by viewing data at its lowest possible unit of significance, the data element. To facilitate the evolution to support this concept the DD/DS should be designed and organized to revolve around the data element.
2. DD/DS interfaces—The most important interfaces that the DD/DS must support are to users of data and to the DBMS. As the ICDB concept evolves and comes to fruition, the users of data will require different types of support. The user

interface with the DD/DS should be able to accommodate these changing requirements. More critical than this last interface is the necessity to support and interface with the DBMS. This should be accomplished with the software interface detailed in previous sections. What is necessary in general in developing the interfaces is to maintain the perspective of the ICDB concept.

CONCLUSION

The central role that the DD/DS can take in an ICDB environment has been illustrated and emphasized. The contention has been made that the first step toward realizing the benefits of the ICDB concept is to install a DD/DS. It remains now for the vendors of data base software to take the initiative and develop a DD/DS that will meet these requirements and thus assist the owners of data (the corporations) in utilizing this resource to its fullest extent.

ACKNOWLEDGMENT

After the manuscript was completed, it was read by Miss Donnie Simons, who helped to clarify thought and improve language.

NOTE

Subsequent to the preparation of this paper, and prior to its publication, IBM announced an Installed User Program (IUP) product entitled "Data Dictionary/Directory". Further information is available in the IBM flier G320-1521-0.

REFERENCES

1 CODASYL SYSTEMS COMMITTEE
Data base task group report
April 1971

2 GUIDE SHARE
Requirements for a data base management system
November 11 1971

3 CONSTRUCTION MANAGEMENT SYSTEM ACTION GROUP
Data base management system requirements
June 23 1971

4 K T SPENCER
Data validation system
Proceedings of GUIDE 33 November 1971

5 H N LIU
A file management system for a large corporate information system data bank
FJCC Proceedings 1968

6 THE DIEBOLD GROUP INC
Organizing for data base management
Doc No 516 December 1971

7 R L ACKOFF
Management misinformation systems
Management Science December 1967

8 R V HEAD
The elusive MIS
Datamation September 1 1970

9 W OLLE
The large data base—its organization and user interface
Data Base Fall 1969

10 D P MOEHRKE
Evolution of data base management at A O Smith
Proceedings of SIGFIDET Workship November 1971

Framework and initial phases for computer performance improvement

by T. E. BELL, B. W. BOEHM, and R. A. WATSON

The Rand Corporation
Santa Monica, California

INTRODUCTION

Computer performance analysis often evokes an image of a hardware monitor dictating a particular hardware modification that doubles the system's capacity. In fact, it usually involves measuring system performance, but is not necessarily limited to the use of hardware monitors, nor does it necessarily involve a hardware modification. It also includes the use of such measurement data sources as software monitors, computer accounting systems, sign-in logs, maintenance logs, and observations from computer operators, system programmers, and users. No specific improvement modification (hardware, etc.) is dictated by the measurements; the analyst must (1) formulate hypotheses about possible inefficiencies and/or bottlenecks in the system by gathering and analyzing computer performance data and (2) suggest alternative system modifications that will improve performance. Such modifications may deal with computer hardware, but they may also deal with computer software, operational procedures, job scheduling, job costing, and any system elements that directly or indirectly affect total system performance.

We suggest a hydrodynamic analogy as one way of visualizing the alternatives open to the analyst who is seeking to improve his system's performance. Consider the computer system as a channel of certain capacity through which a workload is flowing, retarded by several obstructions in the channel, as in the left half of Figure 1. Now suppose that the workload builds and threatens to overflow the channel, as in the right half of Figure 1.

In such a situation, three possible actions can keep the workload from overloading the system.

1. Enlarge the channel, as in Figure 2a, by upgrading the computer to a more powerful model.
2. Remove obstructions from the channel to increase channel capacity, as in Figure 2b. In computer systems, this method involves making modifications in computer system hardware, software, and/or operating procedures to increase system efficiency; it is often referred to as "tuning" a computer system.
3. Reduce the workload flow, as in Figure 2c.

Each method has an appropriate context for application, e.g., if one has a strong requirement for processing a larger, efficiently programmed, compute-bound workload, then upgrading (2a) is necessary because tuning a compute-bound system does not usually make an appreciable difference. Often, however, tuning can increase the flow of an I/O-bound workload enough to meet the new requirements with the existing system (2b). When computing is offered to users as a free good,* situations exist where the workload can be decreased by (1) instituting a direct-charging algorithm or a compute-time rationing scheme, or (2) analyzing application programs to increase their efficiency (2c).

Original (or planned)
Workload
Increased (or actual)
Workload

Figure 1—Hydrodynamic analogy: Situation

These points are simple, but unfortunately not obvious in practice. There is a tendency to settle hastily

* A "free good" is a resource that has no cost to the user and is unlimited in availability.

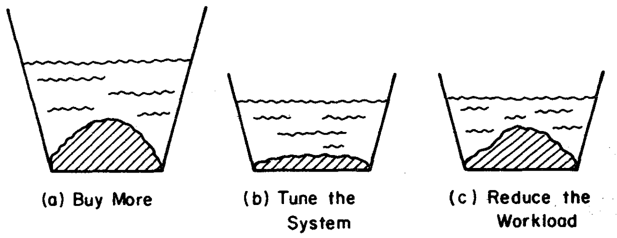


Figure 2—Hydrodynamic analogy: Solutions

on either (2a) or (2b), without much consideration of alternative (2c). These cases result in either a nonsolution that is at best technically interesting to computer system tuners, or an expensive overkill. The objective of a computer performance improvement effort is to determine which areas will improve cost/performance and then achieve the improvement.

OVERALL PROCEDURE

Common performance improvement procedures

After an installation decides that it should be concerned with performance improvement, the most common step is to examine available software and hardware monitors. Salesmen present their products, and one is selected. It is procured and personnel are assigned to begin measuring the system with the unfamiliar tool. This is very expensive; monitors tend to be costly and personnel must be diverted from other work to the new activity. To make matters worse, the measuring process usually severely disrupts machine operation. The payoff for the expense is anticipated in the implementation of a system modification leading to improved performance. Unfortunately, this reward seldom arrives. Instead, the procedure resembles the flowchart of Figure 3. Rand initially tried this procedure; but like most installations that have remained in the field, we modified our approach so that we could be effective in our environment. Our work with other installations indicated that this modified approach could be generalized into an efficient overall procedure, and that procedure has since been effectively applied several times by personnel from both Rand and other installations.

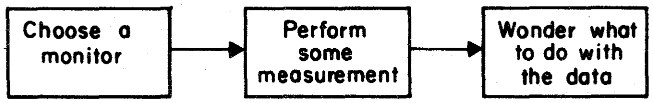


Figure 3—Common procedure

Suggested procedure

The following phased procedure is suggested as an efficient and effective approach toward improving the performance of a computer system. The procedure, summarized in Figure 4, is discussed below.

Understand the system (phase 1)

The first phase of a performance improvement effort involves understanding the particular computer system in terms of management organization of the installation, characteristics of the workloads processed by the computer system, descriptions of the hardware configuration(s) and software programs in use, and information as to what computer-usage data are collected.

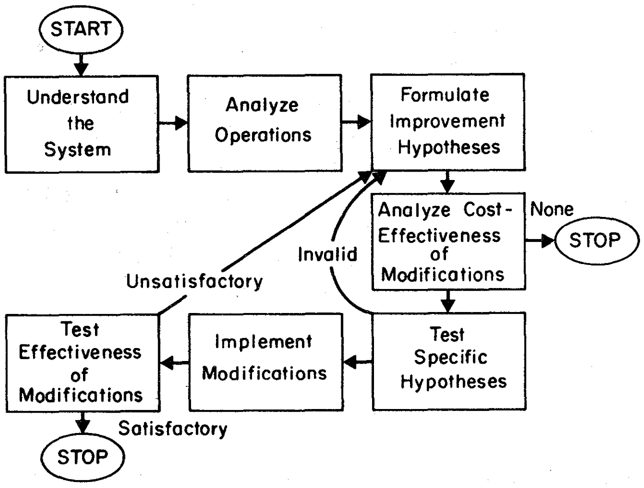


Figure 4—Suggested performance improvement procedure

Analyze operations (phase 2)

The second phase involves the collection of more detailed data to analyze operations. These data, more quantitative than the data collected in the initial phase, provide an analyst with sufficient information to analyze and evaluate the performance at most computer installations. In addition to analyzing operations, data collected in this phase can be useful in reviewing the operational objectives of the installation. Such objectives may include rapid on-line response, low costs, flexibility, easy-to-use software, and good batch turnaround. Because installation management may not have well-defined operational objectives, the analyst must define a set and enumerate their characteristics to assess the criticalness of inefficiencies and/or bottlenecks that are indicated by this analysis.

Formulate performance improvement hypotheses (phase 3)

Based upon system inefficiencies and/or bottlenecks indicated in the analysis of operations (phase 2), hypotheses about probable problems and possible cures can be formulated. These should be specific and performance-oriented. Not

> "It looks like we are I/O bound."

but

> "Our CPU utilization is low, and the disk arms appear to be perpetually jerking around. Perhaps if we reorganize the locations of files on the disk, we will lose less CPU time from programs waiting for the disk arm."

or

> "Our CPU utilization is low, but there does not seem to be much disk-arm or channel contention. Perhaps if we add more core, we could get a third job into execution to soak up those CPU cycles lost during other jobs' I/O operations."

Analyze probable cost-effectiveness of improvement modifications (phase 4)

Before hastily gathering data to test a hypothesis, it is important to analyze whether the resulting performance improvement would be worth the investment. For example, consider the possible hypothesis:

> "If we add another $60,000 worth of communications equipment, we can probably reduce response time from 2 sec to 0.1 sec on our job-query terminals."

Because a two-second response is probably acceptable to practically all terminal users, it would be difficult to justify the cost-effectiveness of such a hypothesis.

Test specific hypotheses (phase 5)

Although short studies (two or three days) usually devote little time to this phase, the bulk of an extensive performance improvement effort is spent here. Figure 5 suggests an ordered set of steps (including feedback loops) to be performed in this phase of performance improvement effort.*

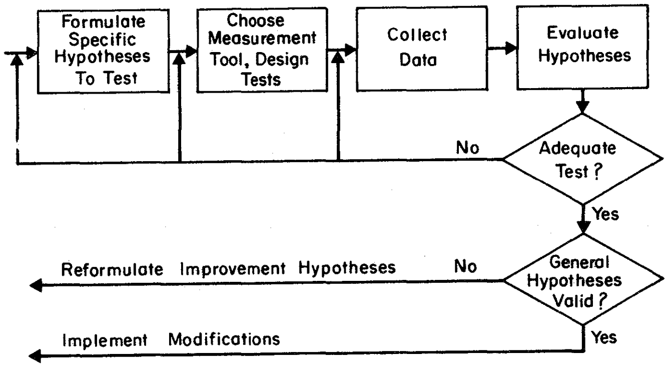

Formulate Specific Hypotheses To Test
Choose Measurement Tool, Design Tests
Collect Data
Evaluate Hypotheses
Adequate Test ?
No
Yes
General Hypotheses Valid ?
Reformulate Improvement Hypotheses
No
Implement Modifications
Yes

Figure 5—Suggested hypothesis testing procedure

Implement appropriate combinations of modifications (phase 6)

Several modifications are often simultaneously implemented because the effort required may be about the same as the effort for only one. Care must be taken that an installation can stand multiple changes without undue impact on production. Also, additional care must be exercised so that modifications do not cancel each other.

Test effectiveness of modifications (phase 7)

Utilizing the measurement tools, data-collection techniques, and test designs used above in "Test specific hypotheses" (phase 5), the effects of modifications upon performance must then be tested. Modifications may result in satisfactory improvements in performance, but often further modifications are necessary to achieve the desired effect. A recycling of the process (starting in phase 3 above) will be required.

Initial phases

Although the suggested procedure includes all phases of the performance improvement effort, the remainder of this paper considers only the first three phases—understanding the system, analyzing operations, and formulating performance improvement hypotheses. We discuss only the initial phases for two

* For material on selection of measurement tools, see References 1–5; for evaluating hypotheses, see Reference 6, which explains the use of accounting data.

reasons. First, the work required in these phases can be generalized and applied to virtually all computer installations—without respect to the type of computer system used or the types of workloads processed. Second, experience has shown these first phases to be frequently slighted by crews eager to try a new measurement device. In contrast, later phases of the performance improvement effort cannot be so generalized. They depend not only upon the results of the first phases, but also on the particular types and configurations of the computer systems studied and the characteristics of the workloads processed.

PHASE ONE—UNDERSTANDING THE SYSTEM

Initial data gathering is very important—whether a performance improvement effort is conducted by installation personnel or by outsiders. Because an analyst from outside the installation is not familiar with its objectives, workload, configuration, etc., he needs the data to identify and work on the correct problem. An analyst from within the installation also needs to collect the data because his background knowledge about some of the specifics is often inaccurate. The following series of general questions include most of the additional data requirements for an outside analyst.

These question have been assembled for the purpose of providing general descriptive information about a computer installation. The questions should provide an analyst with sufficient information about the computer system to initiate a preliminary study in anticipation of performance evaluation and improvement. The information requested by the questions below is usually immediately available at every installation, and should be gathered and recorded at the outset of a performance improvement effort.

These questions comprise a "preliminary questionnaire", to be distinguished from the "detailed questionnaire" to be discussed in phase 2. The information requested is grouped into five categories: organization, workload, hardware, software, and accounting.

Organization

Organizational information is necessary to provide insight into why an installation is interested in analyzing the performance of its computer system(s). This information is also necessary to answer such questions as "who answers to whom?" "who makes what decisions?" "whose approval is required for what?" and questions related to the organization's operational procedures. Questions (1) through (6) should aid in determining this environment.

1. What are the computer center's operational objectives (e.g., providing an on-line system with one-second response time for text editing and two-hour turnaround time on batch debugging jobs)?
2. What situation(s) has provoked a performance improvement effort (e.g., users complaining about poor response time for time-sharing jobs, backlogs of work building up, budget cuts)?
3. What is the organizational position of the computer center with respect to the organization that it serves (e.g., an organization chart)?
4. How are the managers, programmers, operators, etc., structured within the computer center (e.g., an organization chart)? How many systems programmers, applications programmers, operators, etc., are associated with the computer center?
5. Is the computer center run under a "closed shop" or "open shop" philosophy regarding programming, or is there a mixture?
6. How many hours per day and week is the computer system(s) operational (e.g., two shifts per day during the week, and one shift on Saturday)? How many operators per shift?

Answers to the above questions are imperative if an analyst's work is to be relevant to the particular installation. The answers largely define the areas in which alternatives can be considered and, from the stated objectives, indicate where effort should be concentrated. In addition, an "outside" analyst needs this information to determine with whom to speak on his first visit; implied responsibilities can aid in interpreting the comments of in-house personnel.

Workload

Workload information gives insight into some computer system requirements and constraints. These constraints concern the types of jobs that must be processed (e.g., installations that perform much of tape processing or other I/O-bound work often are restricted to low CPU utilization). It is important to know the types and quantities of work submitted to the computer to assess performance in context with any idle capacity or backlog. Also, it is important to determine how jobs are scheduled and priorities assigned. This information gives some idea of how much latitude system personnel have in scheduling for efficient job mix. Questions (7) through (14) pertain to establishing the workload at the computer center.

7. Have any job classes been established to classify and group the workload?
8. What specific projects, programs, and personnel use or request the services of the computer center? (Please include an approximate breakdown of the number of jobs and related machine time submitted to the computer center from the above sources.)
9. Approximately what is the load in terms of the number of jobs on each computer system at the computer center (e.g., 10 percent CPU-bound batch, 25 percent real-time, 40 percent code check, and 25 percent I/O-bound batch)?
10. Does an established schedule exist for production jobs? If so, state times during which the system is dedicated to production runs, and state when it is available for other work.
11. How are priorities assigned to jobs?
12. What have been the general historical trends in workload (i.e., job count, CPU time usage, I/O usage)?
13. Does a backlog ever build up, or does a backlog permanently exist? (Please explain in terms of the number of hours, the priority, and the manner in which the backlogged work is scheduled and processed.)
14. Are there periods during which the computer system is idle? (Please explain in terms of the number of minutes or hours per day/week, and the time during the day/week that idle periods occur.)

Hardware

Information relating to hardware configuration is very useful in gaining a sense of familiarity with the computer system. The outsider's need for this familiarity is obvious, but installation personnel often are not familiar with the details of the configuration either. Knowledge of the make and model of the computer system(s) is therefore also beneficial because analysts involved with performance improvement can review system specifications and capabilities from literature. Information indicative of tuning already performed at the computer center can be obtained through questions relating to hardware modifications. Questions (15) through (17) request information regarding hardware description.

15. What computer system(s) is operated by the computer center? (Plesae supply general hardware configurations for each system.)
16. On what dates have major modifications in hardware configuration taken place, and what were the modifications (e.g., IBM 360/50 to IBM 360/65, additional core added)?
17. What additional hardware components are contemplated or needed at the present time?

Software

The bulk of desirable information regarding software requires much time from the systems programmers, and should be collected in the next phase. General information, though, is necessary in gaining an understanding of the system. Such information is requested in questions (18) and (19).

18. What computer software is used at the computer center? (What operating system is used and what version? What language processors are available and what is an approximate percentage breakdown of use by the programs? What major utility programs are available from the vendor and what utility programs have been written in-house?)
19. What additional software programs are needed at the present time? What software is contemplated for future purchase? What software is currently being developed in-house?

Accounting

Data generated by the accounting system of a computer system may be the most important source of information for assessing performance (unless a software or hardware monitor is available). It is desirable to know what data are recorded by the accounting system and how such data are used. Eventually, the accounting log data may be analyzed to measure and evaluate the performance of the computer system, but for the preliminary study, only questions (20) and (21) need be answered.

20. How is computer usage reported? (That is, assuming an accounting log of some type is generated by the system, what measures of computer usage are recorded?)
21. Is the computer a "free good" or are the users charged for computer usage? If users are charged, what charging algorithm(s) is used? If the

computer is a "free good", is usage budgeted to/by department, projects, etc.?

PHASE TWO—ANALYZING OPERATIONS

In addition to the information supplied in response to the "preliminary questionnaire" (phase 1), more detailed data are necessary to analyze computer system performance. These data are less descriptive in nature and largely quantitative.

At some installations, these detailed data exist only as "feelings" that managers have for their system's performance. Unfortunately, these feelings are not very accessible to others because they exist only in the minds of the managers. The press of day-to-day problems can easily distort such intuitive data to be more reflective of odd situations than normal ones. Even if an accurate feel is obtained, a manager will have difficulty maintaining its accuracy because of the ever-changing nature of the environment.

An organized list of data-collection tasks from which an appropriate subset (for a particular installation) can be taken is presented. The most cost-effective set of data collection tasks for a particular installation may include some not taken from the suggested list. However, the list provides an example of the detail of data collection required. The data collection tasks sometimes appear more onerous than they are. However, they do require a significant effort, which should be justified.

These data-collection tasks are structured in the form of a "detailed questionnaire", and deal specifically with the topics of operational, system, and job characteristics, as well as current measurement and evaluation activities at an installation.

The information requested by the questionnaire should be available either in the form of (1) direct answers from qualified personnel at the installation, or (2) summaries of accounting records and/or reports on computer usage, computer operations, etc.

Data of these types are necessary to analysts working on the performance improvement effort. However, the data are also necessary to installation management for control and improvement of operations on a continuing basis. Because of this latter need, quantitative data of these types should be maintained in an up-to-date manner in order to continuously re-evaluate earlier decisions.

A "detailed questionnaire" is presented below. It does not attempt to ask for all possible information regarding system performance; however, it is exhaustive enough that an appropriate subset should provide sufficient information to analyze and evaluate the performance at most computer installations.

Operational characteristics

1. Determine (from interviewing *operators*) what operational task is considered the most important in maximizing throughput, responsiveness, and/or efficiency of the computer system (e.g., keeping the largest possible number of jobs in core, maintaining control over cards, keeping tapes hung).
2. List any peculiar or extraordinary actions taken by operators to increase throughput.
3. Determine (from interviewing the *head of operations*) how critical good operators are to throughput.
4. Determine (from interviewing the *head of operations*) what the most common operator errors are, and what effect they have on system performance.
5. Determine (from interviewing *operators*) the extent to which the following are done:
   (a) Detailed scheduling of jobs;
   (b) Pre-mounting of tapes and/or disks;
   (c) Talking to users;
   (d) Handling many cards;
   (e) Responding to commands from the console;
   (f) Documenting problems noted by operators.
6. Determine (from interviewing the *head of operations*) how much influence on-line jobs have on operations and throughput.
7. Determine (from interviewing a few *major users*) how adequate turnaround is for batch jobs and how adequate response time is for on-line jobs. Obtain specific estimates.
8. State the number of times the operating system is loaded on an average day.
9. What average amount of time does it take from the beginning of a dead-start until normal operations are under way?
10. State the number of hours that the system is in each of the following states for an average week:
    (a) System operational in native mode;
    (b) System operational in emulation mode;
    (c) System operational in simulation mode;
    (d) System idle;
    (e) System undergoing preventive maintenance;
    (f) System down—hardware failure;
    (g) System down—software failure.
11. State how classified or sensitive jobs are scheduled.

12. State the number of different output forms that are used, and how often the forms are changed during the day.
13. State the number of extra, high-priority jobs that require special scheduling each day. State approximate computer resources used by each of these jobs.

System characteristics

1. Generate a histogram of CPU utilization* over a two-day span, with each period approximately one hour long.
2. State what CPU utilization is averaged over a month, for each of the last two months.
3. Generate histograms of the average number of jobs in core and the number of I/O operations performed per second (or I/O second) over a two-day span, with each period approximately one hour long.
4. State how much use is made of disk storage as compared to tape storage for:
   (a) Temporary (scratch) files;
   (b) Program storage;
   (c) Short-term data storage;
   (d) Long-term data storage;
   (e) Working storage when doing sorts.
5. Give a diagram of the hardware and its interconnections for each computer system in use.
6. Draw core maps.** Or, for any system on which this is not possible (because of relocation or other problems), give a chart showing on-time and off-time for each job and/or job step. The core maps (or chart) should cover a two-hour period at morning start-up, and a two-hour period during the middle of the day. Include (for each job on the core map) the CPU time, the number of I/Os (or I/O time), and the core requested and/or used.
7. Determine (from interviewing the *chief systems programmer*) what appears to be the chief limiting resource (bottleneck) in obtaining more throughput (e.g., core, disk space).
8. Determine (from interviewing the *chief systems programmer*) what elements of the system appear least used.
9. Determine (from interviewing *individual systems programmers*) how much of their time is spent on each of the following efforts:
   (a) Solving system catastrophies caused by the operating system;
   (b) Aiding users to get jobs in the system and to interpret abnormal output;
   (c) Maintaining old applications programs;
   (d) Maintaining and developing new applications programs;
   (e) Maintaining old on-line systems;
   (f) Maintaining and developing new on-line systems;
   (g) Altering software to improve throughput *without* adding new capabilities;
   (h) Responding to changes suggested by the manufacturer;
   (i) Other (please specify)

Job characteristics

1. Generate two frequency charts of CPU usage per job from two months' data—generate one chart for jobs run during prime shift and one for jobs run during off-prime shifts.
2. Generate a frequency chart of the number of tapes used per job, and another for the number of disk files used per job (both over a week's time period).
3. State the percentage of jobs that perform a compile.
4. Generate a chart over a typical time period with the following data for each language used: number of jobs run, CPU time used, and the number of I/Os (or I/O time).
5. State what portion of jobs are classified or sensitive.
6. State the percentage of total CPU time used by production jobs.
7. State the required response time for real-time or on-line inputs.
8. Generate a chart of the 10 or 15 largest jobs (with respect to CPU seconds used) processed during the prime shift of each day in a given week. Generate a similar chart for the off-prime shifts for each day in the same week. For each job, give the elapsed time on the system, and the I/O usage.
9. Generate similar charts requested in (8), but generate them for the 10 to 15 largest jobs with respect to I/O usage.
10. State the inter-arrival characteristics for real-time or on-line inputs (number per hour, etc.).

* The appendix provides an example of such a histogram.
** The appendix provides an example of a core map.

11. State the reasons for which jobs are being run on real-time or on-line systems (instead of batch). That is, why is the quick response needed?

Current measurement and evaluation activities

1. State how data from the following sources are used to evaluate performance:
   (a) Accounting data;
   (b) Hardware monitors;
   (c) Software monitors;
   (d) Other sources.
2. State the exception reporting used to identify abnormal jobs (e.g., a list of jobs requesting more core than allowable, console logs, trouble lists).
3. State how software and/or hardware modifications are evaluated for their effect on system performance.
4. List all internal documentation that exists regarding system modifications and related effects.
5. What specific data obtained from either the accounting log (or facsimile) are used to indicate system performance?
6. What summary reports (daily or monthly) are produced from the system accounting log?
7. List the techniques employed to help improve user/program efficiency.

Adapting tasks to a particular installation

Because different computer installations use different computer systems, operate under different system and operational procedures, and process different types of workloads, modifications to the above tasks may be required when using such a "questionnaire approach" for a particular computer installation.

In practice, we suggest that the detailed questionnaire not be used until the information implied in the preliminary questionnaire is known. The analyst can then use this information to modify some detailed questions after picking the ones that apply to the particular installation. Such modifications might include the deletion, addition, or modification of some questions. An illustration of such modifications is given in the following example:

Suppose that an installation uses an IBM 360 series computer with multiprogramming. In such a situation, task 8 under Operational Characteristics (State the number of times the operating system is loaded on an average day) would be modified to give information about the number of times the operating system had to be reloaded, as opposed to the number of times that the job queue had to be reloaded in addition to reloading the operating system. This question would be modified to say: State the number of "warm start IPLs"* and "cold start IPLs" on an average day.

The detailed information obtained in this phase is very important in the performance improvement effort; without it, hypotheses about how to increase performance become stylized statements only infrequently relevant to specific conditions. Because changes in a system and its load are continuous, data must be constantly revised if a high level of performance is to be maintained in this dynamic environment.

PHASE III—FORMULATING PERFORMANCE IMPROVEMENT HYPOTHESES

The procedure outlined above indicates that hypothesis formulation should follow analyzing operations. This section suggests methods of analysis and describes some general hypotheses that might be appropriate in particular problem situations.

Analysis

The transition from detailed reports to hypotheses is the most difficult phase of the improvement effort. The items in the "detailed questionnaire" are designed to provide reports for this phase. The analyst may use any of several approaches to generate hypotheses, examples of which are given later. Of the five possible approaches explained below, none is the "Eureka" technique, in which the analyst simply stares at the data until suddenly, some basic truth becomes apparent and he shouts "Eureka, I have found it!" This technique is appealing, but the five listed below have been successfully applied far more consistently.

Similar situation identification

The route to a valid hypothesis leading to system improvement is usually marked by ill-formed and incorrect hypotheses and false assumptions. The analyst who has generated successful hypotheses in the past has an advantage because he understands the appropriate detail in which to state hypotheses. In addition, his experience enables him to pursue those hypotheses most likely to be true in his particular

* Initial Program Load.

situation. If the similarity between the present and former situations happens to be great, the analyst need only note the similarity to regenerate the old, successfully applied hypotheses. After enough diverse experiences, the analyst may be able to immediately specify a small number of hypotheses, with a very high probability of being correct on one or more.

This approach for generating hypotheses is most fruitfully applied by analysts with extensive experience. However, even the novice may be able to use it by obtaining his experience vicariously through reading other analysts' reports and by attending professional meetings at which such experiences are presented.

Outlying value discovery

The frequency distribution of job CPU time is usually very skewed, with a long tail containing a few high values. These values appear insignificant in comparison with the many instances of "typical" values. This can easily deceive an analyst into believing that the data are uninteresting, and that other facets of a system should be examined. In many cases, the analyst will have been led away from an important system characteristic; he will have neglected to examine the few, important outliers.

Those few jobs at the tail represent only a small subset of jobs submitted, but they are exactly the ones deserving special attention. In many cases, we have found that less than 5 percent of jobs use more than 30 percent of a system resource. This is very fortunate because only a very small, manageable population need be considered in generating hypotheses.

The above example is only one case in which outliers need special attention for hypothesis generation. Any outlying value—CPU utilization, residency in core system downtime, I/O distribution, or other performance indicator—should be ignored only after it is thoroughly explained. Any residual uncertainty is an indicator that important hypotheses may exist. The analysis leading to uncertainty resolution about outliers can be one of the most successful analysis techniques because outliers are so easy to discover.

Pattern detection

Detecting a pattern in a performance variable over time is more difficult than simply discovering that a group of values is distinctly different from the rest. Common types of patterns in computer performance analysis are cyclical and trend behavior.

Plotting is a powerful technique because it allows the analyst to quickly grasp a large mass of seemingly unrelated data. The key to successful graphing is the selection of both a time scale and, through hypothesis, the data points to plot against them.

Long time periods may be useful when examining workload fluctuations. For example, many computer shops have a periodic workload (in the form of routine reports) that must be produced for normal control in the organization. If this behavior is suspected, then plots of backlog and CPU utilization by day of the week for several months should show a sawtooth form, with the utilization rising a day or so before the backlog. Once it has occurred, the backlog may continue to rise until the weekend allows the shop to catch up—when the process cycles once again.

A much shorter period may be appropriate in other investigations. After using a hardware monitor, one investigator reduced his raw data by taking a computer-driven plotter and preparing time-sequence plots of millisecond time intervals for some of his data. He found the inter-job time period interesting. After plotting several hundred intervals, he could show that the operating system inefficiently used resources between jobs. This, in turn, led him to examine disk assignments, the static contents of core memory set aside to hold system subroutines, and the accounting and charging algorithm, which needed a constant to account for the five seconds lost between jobs to keep track of all wall-clock time when the machine was saturated.

Although many statistical techniques can be used to analyze a time series, these are more appropriate for hypothesis testing than analysis for hypothesis generation. In general, the objective in detecting patterns in computer performance analysis is to generate hypotheses about the causes of patterns rather than quantifying characteristics. The analyst should be concerned with detecting important patterns, hypothesizing causes for them, and determining the validity of the hypotheses.

Although all outlying values should be explained, the analyst cannot expect to understand all patterns because some may be caused by randomness over a short period. However, distinct patterns over any time period should usually be either explained or examined over an expanded data base.

Correlation detection

Detecting a pattern in a single variable is often done merely by plotting the variable's value over time. Detecting correlations between two or more variables is a more formal process in which the analyst must be checking for their existence. This usually occurs after the analyst suspects some correlation, checks its exact nature, and then generates hypotheses that might explain the precise relationship.

Although the existence of a significant correlation may lead to important hypotheses, analysts often have a tendency to assume that a correlation implies a cause-effect relationship. Because correlations can be indirectly caused—or simply spurious—analysis of the correlation's source must be performed very carefully.

For example, there may be a strong positive correlation between a job's memory requirements and its turnaround time. The analyst might therefore hypothesize that the system has inadequate memory. However, the cause might simply be a scheduling algorithm that discriminates against jobs with large memory needs. The system might, in fact, be processor bound, but the detected correlation could lead the analyst away from the fact.

Because correlation detection is one of the more difficult analysis techniques, it should be used with discretion. Other approaches should be attempted first because nearly all hypotheses can be generated through other analysis techniques.

Inconsistency identification and resolution

The identification and resolution of inconsistencies in raw data are conceptually very simple, but this technique is less widely used than it should be. It appears at the end of this list of increasingly difficult techniques because it is so difficult to convince analysts to use. Although rigorous checking of all relationships in data is the most reliable analysis technique for generating hypotheses, it appears most fruitless because it seems to involve verifying obvious relationships. These obvious relationships are usually correct; however, thè few incorrect ones are clues to hypotheses that improve understanding of a system. Adding the hours allocated daily to each activity may show that an installation has either an underused machine or one that is allocated 110 percent. Comparing machine utilization indicated by hardware monitors, software monitors, accounting systems, and metered hours may show that different definitions for "utilization" are implied, and that an analysis of the differences can aid in generating hypotheses.

A common admonition in test equipment use is "when all else fails, read the instruction manual." A computer system analyst should check relationships between (1) data collected from different sources, (2) job-related and system-related measures, (3) measurements that should be the same, and (4) expected and measured data. When all else fails, identify and resolve inconsistencies between data.

Sample hypotheses

The sample hypotheses below are purposely constructed to apply to a large number of systems—in contrast to a detailed set of specific hypotheses that would need generation in an actual performance improvement effort. The hypotheses are organized around the three basic methods of improvement suggested by the hydrodynamic model in the Introduction: reducing the workload, tuning the existing system, and upgrading the computer system. A number of sample hypotheses are given for each of the three basic methods of improvement; however only the first two for each method are discussed in this paper. (The remaining hypotheses are listed, but not discussed.) Detailed discussions of all sample hypotheses are included in Reference 7.

Reducing the workload

As mentioned in the Introduction, reducing the workload keeps the system from overloading. The following five hypotheses identify problem areas associated with overloaded computer systems and indicate possible solutions.

Hypothesis (1): A Free Good Approach Has Led to Large Demands.

Given a zero price for computing, virtually any computing facility will be saturated. Almost any problem that an individual conjures up and programs is worth the zero cost to run it. Therefore, load submittal is limited only by the generation of code. Because programmer time is always scarce (at least to the programmer), choices between human-debugging time and machine-debugging time are resolved in favor of having the machine do it. This sets the maximum submittal rate considerably above the maximum processing rate, except in situations where machine resources grossly exceed human resources.

Too often, a small number of users constitutes the bulk of resource misuse. This has led many installations, particularly universities, to establish a "ten most wanted list" of users who are the worst offenders.

They are encouraged to improve their code and reduce the number of submissions.

Ideally, a tradeoff analysis should be performed to determine at each instant whether a job is worthwhile and whether more programmer time should be devoted to reducing machine resources required. This "ideal" situation, though, would mean that a vast majority of programmer time would be spent performing tradeoffs instead of programming. An alternative is to perform full-cost pricing of machine resources to keep people aware that a computer is not a free good and to provide information to aid in making informal tradeoff analyses.

Hypothesis (2): Unconverted Workload Causes Inefficient System Use.

The recent introduction of a new type or generation of computing system combined with a step increase in demand may be a particularly strong clue to the analyst. Users may be thinking of the new system as merely an extension of the old one. Techniques from a previous generation's system may be extremely poor when applied to new equipment. Inappropriate use of I/O facilities, memory usage, and instruction mixes may cause gross inefficiencies. Simulation and emulation of an old system on the new one are the most obvious sources of inefficient machine use, but they may be a cost-effective alternative to a wholesale conversion effort.

The new system may allow a repressed workload to appear, but inappropriate machine use is a more common cause of a suddenly increased workload. Hypotheses about programming techniques often require great effort for validation, but their effects may justify the effort.

Hypothesis (3): Unlimited "Rights" Lead to Unlimited Demand.

Hypothesis (4): Unneeded Real-Time Systems Consume Scarce Computer Resources.

Hypothesis (5): Organizational Problems Compromise Efficiency.

Tuning the system

The structure of modern computers makes their performance quite susceptible to a variety of seemingly minor details about hardware, software, load, and operating procedures. Tuning is the process of changing these details to make relatively large improvements in performance. Although the performance increase from a single tuning change is often on the level of one percent, a sequence of such changes can lead to considerable increases in performance.

Hypothesis (1): Apparent Minor Actions by Users are Having Strong Adverse Effects.

Tuning can often be performed by influencing user submission of jobs. This can involve actions more detailed than the gross effects referred to in the first hypothesis under "Reducing the Workload." For example, if users have a tendency to submit heavily I/O-bound jobs, a large increase in system performance is often achieved by informing users that larger blocking factors for I/O can increase throughput on the system. Users are usually quite reasonable; when informed of the effects of easily performed actions, they often change to operate in the desired manner. This change is often accelerated by charging schemes or priorities that penalize bad choices. Similarly, bad effects may be caused by large core requirements. Again, informing users of problems often solves the problems without expending funds for hardware. Because of the large leverage that exists for changing the system through informing users, hypotheses regarding workload characteristics should be considered before others. Information about workload characteristics and interviews with several users often provide adequate information for determining the validity of these hypotheses once they are made.

Hypothesis (2): Scheduling by Shift has Compromised Multiprogramming Capabilities.

Hypotheses regarding external scheduling usually involve the characteristics of the mix of jobs submitted to the machine. In some installations, only test jobs are run during prime shifts. These jobs tend to have heavy I/O activity. At night, the CPU-bound jobs are run, taking up large amounts of core and executing for long periods. The result may be a machine that is inefficient because of I/O boundness during prime shift and CPU boundness in the off-prime shift. Hypotheses regarding this can be checked simply by looking at the CPU and I/O activity per job during the two relevant periods.

Hypothesis (3): Inappropriate Short Sequences of Jobs Compromise Multiprogramming Capabilities.

Hypothesis (4): Inappropriate Static Internal Scheduling Priorities Compromise Efficiency.

Hypothesis (5): Inappropriate Dynamic Internal Scheduling Priorities Compromise Efficiency.

Hypothesis (6): I/O Contention for a Specific Device Slows Processing.

Hypothesis (7): Inadequate Core Restricts I/O Overlap.

Hypothesis (8): Inappropriate Operating System Options Reduce System Performance.

Hypothesis (9): Insufficient Peripheral Capability Causes a Bottleneck.

Hypothesis (10): Excess Peripherals Increase Cost.

Hypothesis (11): Unplanned System Initializations are Wasting Processor Capabilities.

Hypothesis (12): Special Requirements Cause Initializations that Waste Processor Capabilities.

Hypothesis (13): Operational Problems are Compromising Efficiency.

Upgrading the computer system

We now discuss the major increases in system resources that are involved in upgrading a computer system. The following hypotheses indicate problem areas that imply upgrading the computer system as a primary alternative solution.

Hypothesis (1): Inadequate Software Resources Cause Overuse of Hardware.

Extremely high CPU utilization and a backlog usually indicate that a major acquisition must be undertaken unless requirements can be reduced through a reduction in jobs submitted or through an improvement in coding. Excess I/O capacity, when combined with high CPU utilization, often is an indication that different programming techniques should be employed. Different compilers, languages, or improved code often help avoid costly equipment purchases.

Hypothesis (2): A Computer System is CPU Bound and Contractual Restraints Prevent Upgrading to a Different System.

Where heavy CPU usage is encountered, additional capability can sometimes be obtained economically by upgrading from a uniprocessor to a multiprocessor system without appreciable increments in I/O resources. This option is particularly attractive when the installation, through such contractual restraints as ownership or long-term lease, cannot economically upgrade to a different system but can augment its processor capability through addition of another CPU. Multiprocessors often have only a hypothesized ability to handle the load to which they may be subjected—often a year or two in the future—because other parts of the system may become limiting with the augmentation of the CPU resource. Such hypotheses must be carefully evaluated by predicting trends in load composition and comparing them with supported multiprocessor options and realized performance.

Hypothesis (3): An Odd Category of Jobs is Overloading the System.

Hypothesis (4): Workload Reductions and Tuning Efforts have been Exhausted, and the System is Still Overloaded.

Caveat

The hypotheses may imply that they should all be applied to each installation. Even preliminary consideration of such hypotheses must be considered within the context of the installation. Special conditions specified by higher level management (such as immediate turnaround for certain jobs) often mean that an installation operates in a manner that in other circumstances would be considered totally inappropriate. However, because situations differ and individual modes of management vary from manager to manager, a variety of operational techniques can be employed to achieve operations, at any individual installation, that are totally appropriate to the management of that installation.

For example, one installation was using third-generation equipment in a distinctly second-generation manner. The utilization of the high-powered CPU was below 30 percent and all output went to tape to be printed offline. On the surface, this appeared to be a highly inappropriate way to use the machines, but a deeper analysis indicated that it was appropriate for this particular situation. The environment dictated that huge amounts of printing be done. As a result, the off-line printing of tapes on a smaller machine was justified in order to unload the higher-powered CPU. This CPU, on the other hand, could not be replaced with a slower machine because the data-transfer rate on slower machines was inadequate to handle the massive amounts of data passing between tape and core. Because of the recent acquisition of this third-generation equipment, programs remained in their second-generation form and were slowly being revised to reflect third generation operating characteristics. There was a valid reason for this: the operation of the entire organization depended on reliably providing management with information for which the installation was responsible.

This installation was in a transitory period between second- and third-generation systems and was knowingly sacrificing operating efficiency in order to achieve higher-level management objectives. As a result of this environment, many of the above hypotheses would be inappropriate, and investigating their validity would be a waste of time, probably alienating operating personnel.

The most common indicator of performance is CPU utilization. If CPU utilization exceeds 70 percent, the *prima facie* assumption is that operations are reasonably well tuned. On the other hand, if CPU utilization is below 30 percent, the assumption is that operations are out of control. These rules of thumb are probably wrong as often as they are appropriate. An installation that reports 90 percent CPU utilization may be achieving this goal only by giving unacceptable turnaround to users and by having poorly programmed code running on the machine. On the other hand, an installation with low CPU utilization may be fulfilling management objectives in the only way possible. One of the more common instances of this is in the realm of highly reliable systems that use multiprocessors. Multiprocessors seldom come with slow-speed CPUs; if the reliability of a multiprocessor is needed, the installation must go to a higher-speed CPU.

In this area, rules of thumb must be surrounded with so many restrictions and additional statements that they become virtually useless. Experience in looking at a number of installations is a necessity for using quantitative data to determine "goodness" or "badness" of operation. This experience can be gained through discussion with personnel at other installations or by actually visiting them. Professional organizations often provide a medium for information interchange that can enable an installation to achieve the collective experience necessary to identify and improve its weak spots.

LATER PHASES

Based on the results of the first phases, the analyst can become far more specific in terms of assessing the overall operation of a computer installation and identifying possible problems and/or bottlenecks in system performance. It is then appropriate for him to consider carefully the details of the current situation and proceed to later phases. Because the first two phases (understanding the system and analyzing operations) assume no information, a common procedure is possible for nearly all situations. Because hypothesis formulation is dependent on information obtained in the first two phases, only indicative hypotheses were given. The remainder of the process is so dependent on the first two phases that each phase is virtually a field of its own.

Special tools (e.g., hardware monitors, special software monitors) are often needed to perform certain types of data collection in later phases of the process. When this is the case, an installation may prefer to hire an outside organization to assist it. Even the most competent outsider, however, will usually not be familiar with all the details of a particular installation. In-house personnel should actively participate in a performance improvement effort and not leave decisions on procedures of testing or system modifications to an outsider, particularly a vendor of the installed system.

The objectives of this paper are to introduce a useful framework and to provide specific suggestions for beginning a performance improvement effort. Even the most impressive measurement tool is useless if applied to the wrong problem. The framework and suggestions about the first three phases are presented to help analysts direct their work to fruitful areas in the later phases.

REFERENCES

1 T E BELL
Computer performance analysis: measurement objectives and tools
The Rand Corporation R-584-NASA/PR February 1971

2 P G BOOKMAN B A BROTMAN K L SCHMITT
Use measurement engineering for better system performance
Computer Decisions Vol 4 No 14 April 1972 pp 28–32

3 L E HART G J LIPOVICH
Choosing a system stethoscope
Computer Decisions Vol 3 No 11 November 1971 pp 20–23

4 K W KOLENCE
A software view of measurement tools
Datamation Vol 17 No 1 January 1 1971 pp 32–38

5 D W WARNER
Monitoring: a key to cost efficiency
Datamation Vol 17 No 1 January 1 1971 pp 40–42+

6 R A WATSON
Computer performance analysis: applications of accounting data
The Rand Corporation R-573-NASA/PR May 1971

7 T E BELL B W BOEHM R A WATSON
Computer performance analysis: framework and initial phases for a computer performance improvement effort
The Rand Corporation R-549-PR August 1971

APPENDIX

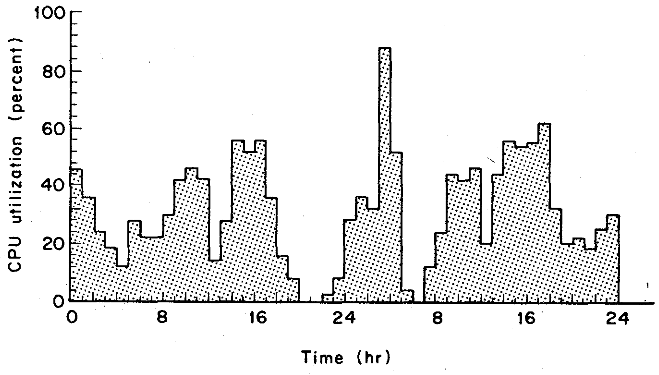


Figure 6—Processor utilization histogram

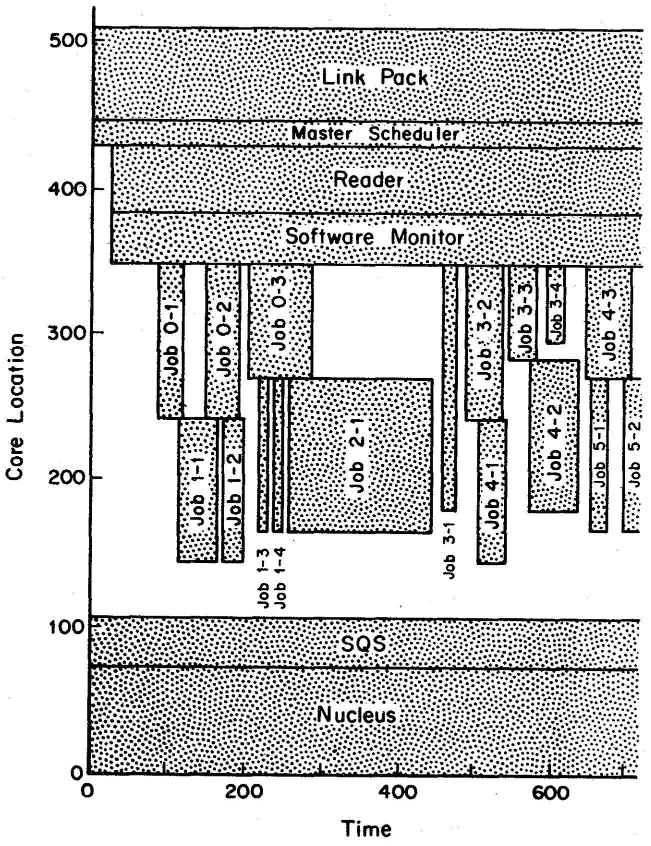


Figure 7—Core map

Core complement policies for memory allocation and analysis*

by STEPHEN R. KIMBLETON

The University of Michigan
Ann Arbor, Michigan

INTRODUCTION

The increasing availability of hardware and software monitors facilitates the measurement of computer systems. The effective utilization and interrelation of these measurements is increased by the development of suitable system models. Model construction, in turn, requires a careful identification of the type of information which the model is to provide.

A primary objective in modeling computer systems is the prediction of system performance as a function of the various policies which may be used to allocate system resources. The two primary resources of a computer system are the CPU(s) and the memory hierarchy (MH). CPU allocation policies have been extensively studied[3,11] as have memory management policies for two level virtual memory systems.[1,5,6,10] However, allocation policies for a multilevel MH having three or more levels have received relatively little attention.[2,15]

This lack of attention would appear to be at least partially due to the fact that although allocation in the first two cases was understood to fall in the domain of the operating system (subject to policy constraints established at system generation time by its designers), control of the allocation of information over a multilevel MH was divided between the operating system and the user. For instance, in most virtual memory computer systems, the operating system controls migration among executable memory (hereafter referred to as core) and the fastest secondary storage device while the user is responsible for controlling the allocation and migration of information among the various remaining levels in the memory hierarchy.

The development of a policy for the efficient allocation and migration of information over all levels in the MH requires knowledge of the "state" of the system in terms of the requirements of the other concurrently executing processes. Since the programmer does not usually have access to this knowledge, it follows that any general solution to this problem must be implemented within the system.

Current policies for migration and allocation of information over an MH with three or more levels are multiple stage policies.[13] That is, the memory management algorithm first migrates information from core to the next fastest level in the memory hierarchy. Provided a block of information remains unreferenced for a sufficiently long time at this level, it is then migrated down to the next level, etc. It is natural to seek a Single Stage Memory Policy (SSMP) in which information is automatically migrated to an appropriate level in the MH upon its ejection from core. Such policies would preclude the need for frequent retransmission of infrequently used information.

In this paper a single stage policy for the allocation of information during the lifetime of a process executing in a paged environment is developed. This policy is shown to be optimal for the case of a single process executing in isolation whose reference string can be characterized in terms of a semi-Markov process. It will be apparent from the development that the results remain valid provided the sequence of return times to each page constitute a sequence of independent and identically distributed random variables. The form of the policy permits a natural extension to the case of multiple concurrently executing processes whose individual reference strings may not be semi-Markov processes.

The results obtained permit comparison of actual MH costs incurred during execution of a process against those which would prevail if the optimal policy were

* This research was partially supported by the Office of Naval Research, Information Systems Program under Contract NOOO-14-67-A-0181-0036 (NR 049-311).

implemented. This allows one to examine the efficiency of the memory migration policy currently being used. Performing this comparison requires the gathering of certain data on system performance which can easily be obtained through the use of a good software monitor.[14]

PROGRAM BEHAVIOR

In Reference 15 the optimal assignment of equally sized information blocks to the MH for a file system is discussed. The necessary extensions to treat unequally sized information blocks are given in Reference 2. The authors appear to be primarily concerned with the techniques necessary for handling rather large blocks of information and therefore feel justified in modeling the reference string of these blocks as an independent trials process. Consequently, they are able to apply techniques related to mathematical programming in order to obtain their results.

Both empirical and theoretical results indicate that the independent trials assumption is untenable when the page is taken as the basic information block.[1,6] Consequently, we are led to the following three assumptions describing the process and its relation to the MH.

ASSUMPTION A1: The page is the basic unit of information transfer and the page reference string (PRS) is an ergodic stationary semi-Markov process (SMP),[16] with transition matrix P and holding time matrix H. The basic unit of time is the interreference time to executable memory.

ASSUMPTION A2: A process has Q pages which can be distributed over $M+1$ levels in the MH labelled $0, \ldots, M$ of which only memory level 0 is executable. T_0 denotes the time between successive references to executable memory and T_j denotes the time required to transfer a page between core and memory level j. The cost per unit time of storing a page at level j is denoted by R_j. We assume $R_0 > R_1 > \cdots > R_m$ and $T_0 < T_1 < \cdots < T_m$.

ASSUMPTION A3: The decision to eject a page from core is made in accordance with a Denning working set policy. Only one copy of a page is maintained in the system and ejection is initiated for a page immediately upon its becoming a candidate for ejection.

Alternatively, one might assume the PRS is a Markov process. However, this requires that the number of consecutive references to a given state must either be identically one or be geometrically distributed. (Let p_{ii} be the probability that the next reference will be to state i given the current reference is to state i. If $p_{ii}=0$, exactly one reference will be made to state i and if $p_{ii}\neq 0$, the distribution of the number of consecutive references to state i will be geometric with parameter p_{ii}.) Since the validity of this assumption is open to question and the added complexity induced by modeling the PRS as an SMP is minimal, we have chosen to do so.

We observe that, in general, the random variables T_j will be dependent upon the amount of traffic between core and a given level j. Since our primary concern is with the development of the structure of an optimal policy, these variations will not be explicitly taken into account. The reader interested in analyzing the effects of queueing properties on the optimal policy will find the approach indicated in Reference 2 to be of use.

The kth cycle for a given fixed page i is the elapsed time between the kth and $(k+1)$st fault for page i. The time of occurrence of these page faults and three other points within a cycle are of interest. In particular, for page i, let $F_k(i)$ denote the time of the kth page fault, let $I_k(i)$ denote the time of the kth injection, let $E_k(i)$ denote the time of initiation of the kth ejection and let $U_k(i)$ denote the time of completion of the kth ejection. To correctly sequence the indices of these random variables we assume that a page does not initially reside in executable memory.

Our primary concern is not with the instant at which an event occurs but rather with the elapsed time between their various occurences as summarized in:

DEFINITION. For a given fixed page i and $k \geq 1$:

(i) $A_k(i) = I_k(i) - F_k(i)$
denotes the time required to retrieve the ith page for the kth time,

(ii) $B_k'(i) = E_k(i) - I_k(i)$
denotes the kth extended core residence time (ECRT) of page i,

(iii) $C_k(i) = U_k(i) - E_k(i)$
denotes the time required for the kth ejection of page i,

(iv) $D_k'(i) = F_{k+1}(i) - U_k(i)$
denotes the kth extended core complement time (ECCT) of page i, i.e., the elapsed time from completion of the kth ejection until the $(k+1)$st page fault occurs for page i.

The random variable $B_k'(i)$ contains several subintervals of time of total length, say, $\beta_k(i)$ corresponding to time expended while page i is in core waiting for a page fault to be satisfied for another page. We let $B_k(i) = B_k'(i) - \beta_k(i)$. Analogously, we let $\delta_k(i)$ denote the total amount of time in $D_k'(i)$ which the page spends on a secondary storage device while waiting for a page fault for some other page to be satisfied, and let $D_k(i) = D_k'(i) - \delta_k(i)$. We refer to $B_k(i)$ and $D_k(i)$ as the core residence time (CRT) and core complement time (CCT) respectively. Further, the level in the memory hierarchy at which the page is stored when not in core will be known as its core complement location (CCL). Observe that the activity of a page during a cycle (exclusive of the time the process spends in page fault) is completely described by its CRT, CCT and CCL. The activity of the page including the time spent in page fault is described by its ECRT, ECCT and CCL.

Let $L_k(i) = A_k(i) + B_k(i) + C_k(i) + D_k(i)$ and let $L_k'(i) = A_k(i) + B_k'(i) + C_k(i) + D_k'(i)$. Denoting the expected value of a random variable X by x, let $\rho = l_k'(i)/(b_k(i) + d_k(i))$. Then ρ represents the delay per unit time induced by executing the process in a multilevel environment instead of with all pages in core. Using standard techniques, it is easy to verify that the ergodicity of the underlying SMP implies ρ is independent of i. For later use we also note that since $A_k(i) + C_k(i) \ll B_k(i) + D_k(i)$ in any reasonably managed memory environment it is reasonable to assume $\rho = l_k'(i)/l_k(i)$.

Since ρ is a measure of the extent to which process execution is delayed by operating in a multilevel MH, a normal design objective is to ensure ρ is near one by minimizing the amount of time the process spends in the page wait state. However, for business data processing installations with very large file handling requirements which need only a small amount of processing, ρ may be significantly larger than one.

The preceding comments remain valid for multiple concurrently executing processes. However, the intervals $B_k'(i)$ and $D_k'(i)$ would have subintervals corresponding to the time the processor is servicing other processes.

In the following the sequences of random variables $\{A_k(i);\ i \geq 1\}$ and $\{C_k(i);\ i \geq 1\}$ will be assumed independent and identically distributed. If the assignment policy places pages at the same level as that from which they were retrieved, it is also reasonable to assume that $\{A_k(i)\}$ and $\{C_k(i)\}$ possess the same distribution.

The net effect of our assumption that the PRS is an SMP is provided by the following theorem. Consequently, this theorem exposes the main limitation of the model which in effect requires that the intervals between page faults must constitute a sequence of independent and identically distributed (*iid*) random variables. There is some evidence[6] to indicate that the results of the theorem are a reasonable approximation to reality for programs possessing good locality and a low page fault rate.

Theorem

Under assumptions A1-A3, for a single process executing in isolation, the sequences of random variables $\{B_k(i);\ i \geq 1\}$, $\{B_k'(i);\ k \geq 1\}$, $\{D_k(i);\ i \geq 1\}$, and $\{D_k'(i);\ i \geq 1\}$ are *iid*.

Since we shall not need to refer to a particular cycle in the following, we agree to drop the subscript k. Further, unless otherwise indicated, we shall always be referring to a given fixed page i, and thus we shall not explicitly indicate the dependence of these random variables on i.

In the remainder of the paper usage of the terms CRT, CCT, ECRT and ECCT will refer to their expected value and not an individual sample value.

THE PAGE COST STRUCTURE

The expected cost of a page to the system during a cycle of expected length $l' = a + b' + c + d'$ is the sum of:

(i) $R_j d'$
the cost of storage on memory level j for a length of time d',

(ii) $(R_0 + R_j)(a + c)$
the cost of transmission between core and the secondary storage device,

(iii) $R_0 wa$
the cost of keeping the average working set in core during the time required to satisfy a page fault, and

(iv) $R_0 b$
the core cost of the page during b,

which is:

$$(R_0 + R_j)(a + c) + R_0(wa + b) + R_j d'. \qquad (1)$$

Observe that the cost of the page during the time interval $\beta = b' - b$ is subsumed in the analogue of (iii) for those pages generating the page faults of total length β. Since the PRS is a stationary SMP, the level at which the page is stored when not in core will not vary and thus $a=c=t_j$ if level j is the CCL. Thus (1) may be rewritten as:

$$2(R_0+R_j)t_j+R_0(wt_j+b)+R_j\,d'. \qquad (2)$$

The sequences of random variables $\{L_k\}$ and $\{L_k'\}$ are *iid* since their summands are by the preceding theorem. Consequently, it is reasonable to define the average cost per unit time of a page by:

$$[2(R_0+R_j)t_j+R_0(wt_j+b)+R_j\,d']/l'. \qquad (3)$$

(The mathematically inclined reader might also note that a basic result for renewal reward processes [16, Theorem 3.16] proves that this definition is consistent with the usual interpretation of the term *average reward per unit time.*) Since $d'=\rho d$ and $\rho^{-1}=l/l'$, this expression may be rewritten as either:

$$f_j(d)=\rho^{-1}[2(R_0+R_j)t_j+R_0(wt_j+b)+\rho R_j\,d]/l \qquad (4\text{a})$$

or

$$q_j(d')=\rho^{-1}[2(R_0+R_j)t_j+R_0(wt_j+b)+R_j\,d']/l. \qquad (4\text{b})$$

Given the cost structure described in equations (4) we may now seek an optimal (minimal) cost CCL in terms of b and w which are assumed fixed in the remainder of this section. (In making a choice regarding the use of either (4a) or (4b) observe that allocations made in terms of d' reflect page fault time and are thus interdependent with the assignments made for the other pages. Allocations made through the use of d do not suffer this disadvantage, but the measurement of d is somewhat more difficult.) The optimal policy can be developed in terms of either d or d'; we shall use d. The details for d' are completely analogous and will be left to the reader.

The optimal CCL for a page with CCT d will be at level n provided:

$$q_n(d)\leq q_j(d), \qquad 1\leq j\leq M. \qquad (5)$$

We now prove the optimal policy is a cutpoint policy, i.e., the real line can be subdivided into intervals $I_n[=I_n(b,w)]$, $0\leq n\leq M$ such that $d\in I$ implies CCL n is optimal. This follows, since (prime denotes differentiation) $j<k$ implies $q_j(0)\leq q_k(0)$, $q_j'(d)\geq q_k'(d)$ and $q_j'(d)$, $q_k'(\mathrm{d})<0$. That is, the cost per unit time of a page stored at level j when not in core is monotone decreasing to ρR_j. Since storage at a slower level is more expensive for small CCTs, and since $0>q_j'(d)\geq q_k'(d)$, it follows that there can be at most one positive point of intersection say $x_{jk}[=x_{jk}(b,w)]$. This situation is indicated graphically in Figure 1.

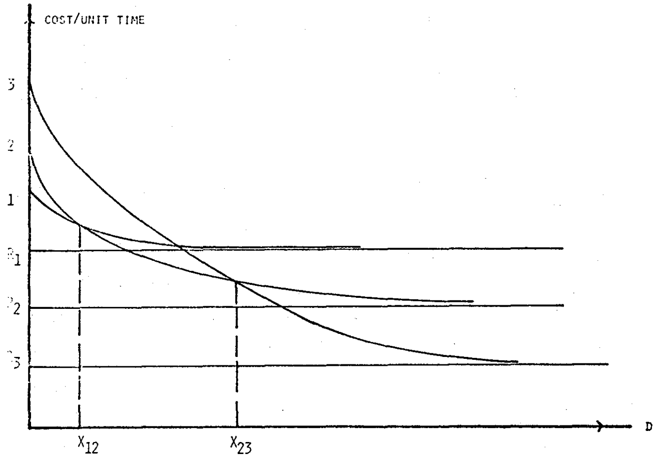


Figure 1—Illustration of behavior of $\mathrm{F}_j'(\mathrm{D})$, $\mathrm{J}=1,2,3$ and points of intersection

It may happen that $q_j(d)$ and $q_k(d)$ have no point of intersection. In this case placement of a page at level k always costs more than placement of the page at level j. Consequently, memory level k may be deleted from further consideration for the given (b,w). Following [15] the set of memory levels on which information should actually be stored will be termed the derived hierarchy. This hierarchy may be obtained in the following manner.

Memory level 0 will always be in the derived hierarchy since it is the only executable level. Assume that it has been established that memory levels $j-1$ and j should be in the derived hierarchy. The next level in this hierarchy corresponds to the first index $k>j$ for which $x_{jk}>x_{j-1,j}$. In the case of ties we agree to use that level with the smallest index since it may be accessed more rapidly.

In the remainder of the paper we shall only be concerned with the derived hierarchy. This being the case, for simplicity in notation, we shall assume that there are M such levels labeled $0,\ldots,M$. This double usage of the symbol M seems reasonable since we shall never need to refer to levels not in the derived hierarchy. For this hierarchy, we have shown that the optimal policy has the following form. The real line is divided into intervals $I_j=[x_{j-1,j},\,x_{j,j+1})$. If $d\in I_j$, the optimal assignment for a page with CCT d, CRT b and working set w is level j. Note that if $d=x_{j,j+1}$ the decision to store the page at level j or $j+1$ is arbitrary. Observe that if the CCL is to be determined from d', it follows from (4b) that storage will be at level j provided $d'\in[\rho x_{j-1,j},\rho x_{j,j+1})$.

It should be noted that our discussions have always assumed that an adequate amount of memory existed at each level. However, if the available memory at a given level has been exhausted, this simply corresponds to removing that level from the derived hierarchy. Thus, assume that levels $j-1$, j and $j+1$ are in the derived hierarchy. An assignment is made to level j if $d \in [x_{j-1,j}, x_{j,j+1}]$. If level j has been exhausted and level $j+1$ is available, the assignment is then made to level $j+1$ since $x_{j-1,j} < x_{j,j+1} < x_{j+1,j+2}$. This stability and the consequent ease of making appropriate level assignments should be compared with the relative complexity of other policies.[2,15]

Finally, let us remember that the push algorithm determines when a page should be ejected from core. If a given page is referenced often enough (i.e., $d < x_{0,1}$) it is possible that it may never be ejected. In that case, clearly, no further determination of the optimal location need be made.

AN OPTIMAL POLICY

Establishment of an optimal policy requires the identification of[17]: (a) a method for judging optimality, (b) the sequence of times at which a decision can be made, (c) the set of decisions which can be made, and (d) the information to be used in making these decisions. We assume these to be average cost per unit time, the sequence of instants at which a page is ejected from executable memory under a working set policy, the memory level to which the page is migrated and d, b, w respectively.

The cost of executing a process on a given computer is the sum of the CPU cost, the MH cost (we assume channels are included in the MH) and the costs of unit record equivalents, e.g., card readers/punches, printers, etc. For a symmetric multiprocessor system possessing an adequate accounting routine, the variations in the cost of using the CPU(s) over different executions of the same job as measured by the total amount of CPU time required by the job should be less than one percent. Further, the costs of unit record equivalents may be billed on a usage basis.[12] Thus, minimization of the system costs is equivalent to minimization of the costs of the memory hierarchy.

To minimize the costs of all the pages of the process over the MH, first observe that the assignment policy for a given page is based only on the properties of that page as measured in terms of CRT, CCT, w and access times. The other pages of the process influence the assignment of a given page only through the expected size of the working set, whose implementation, in turn, requires no comparison of the properties of a given page with those of the other pages in core to determine the eligibility of the page for ejection.

It follows that we may minimize the costs associated with an SSMP by optimally assigning each page individually in accordance with the policy developed earlier. That is, if α is a memory policy, i.e., α is a map of $(d, b, w, \{t_j\}, \{R_j\})$ to a given level in the hierarchy, and $C(\alpha)$ denotes the MH costs associated with policy α for a given process, then:

$$C(\alpha) = \sum_{1}^{Q} C(i, \alpha)$$

where $C(i, \alpha)$ represents the cost of page i to the system under policy α. Consequently our objective is to determine α^* such that:

$$C(\alpha^*) = \underset{\{\alpha\}}{\text{MIN}}\, C(\alpha)$$

However:

$$C(\alpha^*) \geq \sum \text{MIN}\, C(i, \alpha)$$

and the cost on the right hand side can be achieved through application of the policy described in the previous section. Note that these remarks are also valid for the case of multiple concurrently executing processes. However, the optimal policy for a single page was obtained only in the context of a single process executing in isolation.

AN EXAMPLE

In an earlier section we have shown that given the CRT b, the optimal policy is a cutpoint policy. Equation (5) and the discussion following implies that the level change point (LCP) from storage at level j to storage at level $j+1$ for a given b, w is the intersection of $q_j(d)$ and $q_{j+1}(d)$. The LCP's may be easily determined since the resulting equation is a quadratic with one negative root and one positive root.

A short program was written to perform the necessary root determinations and the results of this program are indicated in the following tables. We assumed four levels in the memory hierarchy corresponding to core, drum, disk and datacell. Using the monthly rental figures for the costs of core, 2301 drum, eight drive 2314 and 2321 datacell (400M bytes), the ratio of the costs of core, drum, disk and datacell to core for storing one page of information for one unit of time were found to be 1.0, .015, .0068, and .0019 respectively. We assumed $\rho = 1$ (note that the corresponding tables for $\rho \neq 1$ can be obtained by scaling) and a mean core interreference time of one microsecond, a mean access time to drum of 10 *ms*., to disk of 50 *ms*. and to datacell of 350 *ms*. With these hardware characteristics and $w = 10$, $b = 100$, $d = 1000$, each of the parameters was then varied in-

TABLE I—LCP variation as function of CRT b (ms.)

Level Switch	b=20	b=100	b=1000
0-1	122 ms.	122 ms.	122 ms.
1-2	58.6 s	58.5 s	58.4 s
2-3	735 s	734 s	736 s

TABLE III—LCP variation as function of level 1 access time

Level Switch	T(1)=2.5	T(1)=5.0	T(1)=7.5	T(1)=10
0-1	31 ms.	61 ms.	92 ms.	122 ms.
1-2	69.4 s	65.7 s	62.0 s	58.4 s
2-3	734 s	734 s	734 s	734 s

dividually to examine the sensitivity of the LCP to this variation.

It is apparent from Table I and the preceding remarks that the LCP is reasonably insensitive to variations in *b*. Variations in *w* have a very pronounced effect as do variations in access times or cost ratios.

CONCLUDING REMARKS

The preceding policy has been obtained in the context of a single process executing in isolation. Its use provides a means for judging the efficiency of other allocation policies. The form of this policy permits an easy extension to the case of multiple concurrently executing processes. The mathematical justification for this extension will require additional development since the transitions among the pages of the collection of concurrently executing processes will not be an SMP.

It should be noted that the implementation of this policy does not require detailed measurements to determine the precise form of the transition and holding time matrices. Only the average CRT and CCT (or CRT and ECCT) for each page must be measured or estimated as well as the average working set size of the process. In view of the relative insensitivity of the policy to changes in the CRT as indicated in Table 2, the use of a nominal value for the CRT would be appropriate. Further, an examination of Tables 1-6 indicates that the policy is fairly stable as a function of *d* (i.e., small variations in *d* will not produce a large variation in the level at which a page is stored). Consequently, provided the approximate value of ρ is known, use of the ECRT allows the implementation at the cost of two words per page plus one additional word per process for the estimator of the average working set size. Word one would be used to maintain a moving average for the CCT and word two would be time stamped upon the ejection of a page from core. Upon its return, the current time would be compared with the time stored in word two and the result would be used to update the moving average.

TABLE II—LCP variation as function of working set size w

Level Switch	w=1	w=10	w=100
0-1	31 ms.	122 ms.	1035 ms.
1-2	14.6 s	58.5 s	497.4 s
2-3	183 s	734 s	6244 s

For multiple concurrently executing processes having no shared pages, the effect of the other processes on a page of a given process will be to lengthen its CRT and CCT by an amount of time related to the amount of time that the system devotes to the other processes. The average working set size will remain unchanged. Because of the relative insensitivity of the policy to changes in the CRT it follows that a page may be stored on a slower I/O device than would be indicated by the policy for that process executing in isolation. However, for the present generation of I/O devices, the differences in the characteristics of the various levels in the memory hierarchy are so great that it seems unlikely that this would often be the case.

To analyze the effects of multiple processes on a shared page, note that the CRT, CCT and transmission time of a page may be determined independently of whether or not the PRS is an SMP. Viewed in this light, it is evident that a page shared among multiple concurrently executing processes will experience a larger CRT and smaller CCT than the corresponding quantities of any of the processes executed in isolation. The natural analogue for a shared page of the cardinality of the working set is the cardinality of the union of the working sets for the individual processes which reference that page. Consequently, as would be expected, this would tend to lead to storing the page at a higher level in the memory hierarchy.

TABLE IV—LCP variation as a function of level 2 access time

Level Switch	T(2) =20	T(2) =35	T(2) =50	T(2) =65	T(2) =80
0-1	122 ms.	122 ms.	122 ms.	122 ms.	122 ms.
1-2	14.5 s	36.5 s	58.4 s	80.4 s	102.2 s
2-3	807 s	770 s	734 s	697 s	660 s

TABLE V—LCP variations as a function of level 3 access time

Level Switch	T(3) =200	T(3) =350	T(2) =500	T(3) =650	T(3) =800
0-1	122 ms.	122 ms.	122 ms.	122 ms.	122 ms.
1-2	58.4 s	58.4 s	58.4 s	58.4 s	58.4 s
2-3	366 s	734 s	1100 s	1468 s	1833 s

The assumption that the PRS is an SMP provides: (i) a more realistic assumption than the independent trials assumption, and (ii) a manageable characterization of program behavior which is necessary in order to be able to obtain an optimal policy. The departure from reality is due to the fact that program references tend to exhibit a time-varying character, often referred to as the property of locality.[5,6]

The time variation in the PRS could be conceptually accommodated through the use of a time-varying, i.e., nonstationary SMP. Analysis of a general nonstationary SMP is very difficult. However, the preceding policy could be used to analyze the following simple time-varying case. Assume there are a sequence of time instants $t_1, t_2, \ldots$ and a sequence of semi-Markov processes (P_1, H_1), $(P_2, H_2), \ldots$ such that the PRS is represented by the SMP (P_i, H_i) over the time interval (t_{i-1}, t_i). If $\{t_i\}$ is known, the preceding policy could be applied by changing the measured or estimated values of w, CCT and CRT at these switch points.

Our discussion has tacitly assumed that only the amount of memory needed at any given level must be acquired. It is more realistic to assume that memory must be acquired in modules. Should this be the case, the policy developed in this paper may be used to determine the optimal location of information among memories and the minimum cost system for this optimal location. This cost may then be compared with the actual cost imposed by the modularity of memory to determine the "inefficiencies" caused thereby. Further, because of the previously established stability of the cutpoint policies, it is evident that the most desirable blocks of memory to move into any level with extra space requires consideration of only those blocks with CCT's closest to those of that level.

TABLE VI—LCP variations as a function of cost ratio variations. Ratios are (1, .1, .0068, .0019), (1, .1, .05, .0019) and (1, .1, .05, .025) in columns 1, 2, 3 respectively

Level Switch			
0-1	135 ms.	135 ms.	135 ms.
1-2	5.01 s	9.5 s	9.5 s
2-3	733.9 s	74.0 s	143.2 s

BIBLIOGRAPHY

1 A V AHO P J DENNING J D ULLMAN
Principles of optimal page replacement
J ACM 1970 Vol 18 No 1 pp 80-93

2 S R ARORA A GALLO
The optimal organization of multiprogrammed multilevel memory
Proceedings of the ACM Workshop on System Performance Evaluation
Harvard University April 5-7 1971 pp 104-141

3 E G COFFMAN L KLEINROCK
Computer scheduling methods and their countermeasures
SJCC 1968 pp 11-21

4 P J DENNING
The working set model for program behavior
C ACM 1968 Vol 11 No 5 pp 323-333

5 P J DENNING
Virtual memory
Computing Surveys 1970 Vol 2 No 2 pp 154-189

6 P J DENNING J E SAVAGE J R SPIRN
Models for locality in program behavior
TR No 107 Department of Electrical Engineering
Princeton University Princeton New Jersey May 1972

7 D J HATFIELD J GERALD
Program restructuring for virtual memory
IBM Systems Journal 1971 Vol 10 No 3 pp 178-192

8 R A HOWARD
Dynamic programming and Markov processes
MIT Press 1960

9 D J KUCH D H LAWRIE
The use and performance of memory hierarchies: A survey
In Software Engineering
Vol 1 Academic Press 1970

10 R L MATTSON J GECSEI D R SLUTZ
I L TRAIGER
Evaluation techniques for storage hierarchies
IBM Systems Journal 1970 Vol 2 p 78

11 J M McKINNEY
A survey of analytical time-sharing models
Computing Surveys 1969 Vol 1 No 2 pp 105-116

12 *MTS manual*
Volume 1 1971 The University of Michigan

13 E I ORGANICK
The MULTICS system
MIT Press 1972

14 T B PINKERTON
The MTS data collection facility
Memorandum 18 CONCOMP Project The University of Michigan June 1968

15 C V RAMAMOORTHY K M CHANDY
Optimization of memory hierarchies in multiprogrammed systems
J ACM 1970 Vol 17 No 3 pp 426-445

16 S M ROSS
Applied probability models with optimization applications
Holden-Day San Francisco 1970

17 D TEICHROEW
An introduction to management science: Deterministic models
J Wiley New York 1964

APPENDIX

PROOF OF THEOREM

The following proof is for a given fixed page i and to simplify notation explicit indication of the dependence of the random variables on i will be suppressed. We first verify $\{B_k\}$ and $\{D_k\}$ are *iid.*

Use of a working set policy with window τ implies that a page is core resident for only as long as it is accessed every τ time units during the periods when the process is active.[4] Since the PRS is an SMP the sequence of return times $\{R_n\}$ to page i is *iid.*[16] Further, if:

$$L_n = \begin{cases} R_n, \ R_n \leq \tau \\ 0, \text{ otherwise} \end{cases}$$

$$G_n = \begin{cases} 0, \ R_n > \tau \\ R_n, \text{ otherwise} \end{cases}$$

then $R_n = L_n + G_n$ and $\{L_n\}$ and $\{G_n\}$ are *iid.* However:

$$* \qquad B_k = \sum_{n=1}^{N_k} L_n + \tau$$

where N_k counts the number of returns to page i until a return time exceeding τ is obtained. If $1-\rho$ denotes the probability of a return exceeding τ, it follows that the distribution of N_k is geometric with parameter ρ. Consequently $\{N_k\}$ is *iid,* and (*) thus implies that $\{B_k\}$ is *iid.*

Since $\{G_k\}$, $\{C_k\}$ are *iid* and $D_k = \max\,(0, G_k - C_k)$, it follows that $\{D_k\}$ is *iid.*

Let J_k denote the number of page faults during D_k. Since the PRS is an SMP, $\{D_k\}$ *iid* implies $\{J_k\}$ *iid.* Consequently, $D_k' = D_k + \sum_{n=1}^{J_k} Y_n$, where Y_n represents the time to satisfy a page fault. In view of our assumptions, $\{Y_n\}$ is *iid.* Consequently $\{D_k'\}$ is *iid.* Verification that $\{B_k'\}$ is *iid* is analogous.

Data modeling and analysis for users—A guide to the perplexed

by ARNOLD F. GOODMAN

McDonnell Douglas Astronautics Company
Huntington Beach, California

INTRODUCTION

There are styles in management, science and technology, just as there are styles in politics, music and much of life. At any given time, some concepts and approaches are in—with lots of action surrounding them—and other concepts and approaches are out—with not very much action surrounding them. And now the winds of change are blowing strongly.

Disciplines which have traditionally been less mathematical in their development are becoming more mathematical, with each passing year. Concepts such as mathematical modeling and computer simulation are becoming increasingly popular and useful. Approaches oriented toward solving real problems with real or simulation data are becoming more and more prevalent.

In fact, current management, science and technology are literally characterized by their dependence upon a tremendous amount of complex data which are generated for analysis. Many important resources are expended in data generation, and many significant decisions await data analysis. Data modeling and analysis have become not only a significant topic, but also a stylish one.

That the above statements are true for science and technology is easily verified by an inspection of their literature. That they are also true for management is supported by a recent survey of new uses for computers in business.[1] As for government, there is a current emphasis on cost benefit modeling and analysis within the popular Planning, Programming and Budgeting System (PPBS), the approach of the Environmental Protection Agency, and the trend toward technology assessment and guidance.

Introduction of modeling and analysis into areas where they have not traditionally been employed—in order to raise the level of decision from one of supposition to one of logic—is, however, as delicate and difficult a process as it is essential for success. There are many such introductions which have failed, due to actually lowering the level of decision to one of apparent logic. An insider's view of data modeling and analysis is presented, in order to help guide users through the potentially rewarding—but hazardous and perplexing—maze. An appropriate perspective on the topic is attained by discussing it in terms of its objective, solving problems.

Three areas of phenomenal growth—computer utilization, computer technology and computer science—have produced the requirement for a new discipline, measurement of computer systems. In an atmosphere of escalating computer cost and increasing budget scrutiny, measurement provides a bridge between design promises and operational performance. This function of measurement is complemented by the traditional need for measurement of any art in search of a science.

Measurement of computer systems has been named "compumetrics"—in the spirit of biometrics, econometrics and psychometrics—by Hamming.[2] It concerns measurement in or of computer systems and is extensively discussed by the author.[3] At present, compumetrics might be characterized as a growing collection of measurements on their way toward a science, and in need of planning and analysis to help them get there. Measurement of computer systems provides a specific context in which to view data modeling and analysis.

Bell, Boehm and Watson[4] adapt the scientific method to performance measurement and improvement of a computer system: from understanding the system and analyzing its operation, through formulating performance improvement hypotheses and analyzing the probable cost-effectiveness of the corresponding modifications, to testing specific hypotheses and implementing the appropriate combinations of modifications—as well as testing the cost-effectiveness of these combinations. This paper complements that one by proposing an approach to viewing and utilizing such a framework.

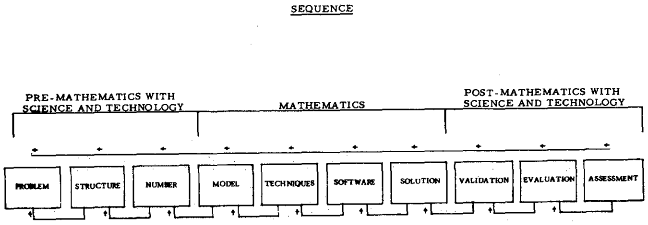
The presentation begins with a discussion of the sequence from a problem through a solution to its assessment. Some observations are then made concerning aspects of solving problems which should be considered—but all too often, unfortunately are not. Finally, an approach is described for the design and analysis of a complex system through utilization of both experimental and computer simulation data.

As is appropriate for a guide to users, comments are general and suggestive, rather than detailed and complete. There are too many treatments of data modeling and analysis which concentrate upon the technical details, at the expense of general philosophy and approach. It is probably true, however, that many more mistakes of a much more serious nature are due to improper general philosophy and approach than are due to incorrect technical details. In addition, words are employed in their usual nontechnical sense.

PROBLEM SOLVING SEQUENCE

In solving problems with mathematics, the sequence of steps which leads from the problem through a solution to its assessment—the problem solving sequence—may be viewed as being composed of pre-mathematics, mathematics and post-mathematics (see Figure 1). Pre-mathematics begins with the problem, develops a structure—or framework—for it, and then associates number with the structure. Mathematics begins with construction of a model—or picture—for the numerical structure, utilizes appropriate techniques and software, and then obtains a solution to the problem. Post-mathematics begins with validation of the solution, evaluates the quality of that solution, and then assesses its impact—or effect. Since the solution's assessment should influence the entire problem solving sequence in reverse—by implying how much blood, sweat and tears should be expended on each portion of the sequence, in turn—it should be considered prior to actually beginning the sequence. For a given problem, the entire problem solving sequence may not be feasible. An illustrative example of pre-mathematics and mathematics is described by the author.[5]

Mathematics has traditionally grown and prospered in conjunction with development of the measurement oriented disciplines in science and technology: where both the pre-mathematics of problem structure and number association, and post-mathematics of solution validation, evaluation and assessment were relatively simple to accomplish—in addition to being relatively unimportant in the problem solving sequence. However, pre-mathematics and post-mathematics are now becoming more important to those management, scientific and technical disciplines which have not been measurement oriented.



ASPECTS

- ●EXPLORATION, EXPLOITATION AND EXPLANATION
- ●PRIMARY VERSUS SECONDARY
- ●PERFECTION VERSUS SUFFICIENCY
- ●ANALYSIS VERSUS SYNTHESIS
- ●SPECIALTY VERSUS GENERALITY
- ●COMMUNICATION WITH COLLEAGUES AND USERS
- ●EFFECTIVENESS VERSUS EFFICIENCY
- ●ACCURACY VERSUS PRECISION
- ●NUMBER ASSOCIATION, UTILIZATION AND INTERPRETATION

Figure 1—Problem solving

A typical problem of significance today and tomorrow—such as measurement of computer software—will be like a large gelatinous mass: we will reach for it in order to develop structure and associate number, but it will squirt out of our hand. We will somehow have to delve beneath the problem's soft exterior and develop a structure for it, and then associate number with that structure. At the other end of the problem solving sequence, we will have to validate, evaluate and assess the solution in the same squishy environment. And both of these will be as delicate and difficult to accomplish as they will be essential for success.

PROBLEM SOLVING ASPECTS

Over the past five years, I have closely observed mathematically oriented scientists and engineers—well educated, well experienced and well paid—approach the solution of software problems involving computer and information systems. There are certain aspects of solving problems with mathematics which we should consider, if we are to increase our probability of success—whether they are chemical or computer problems, ecology or engineering problems, legal or linguistic problems, management or medical problems, physical or planning problems. However, most of these scientists and engineers fail to consider many such aspects—both in doing their own work and in reviewing the work of others. Why?

The approach of those scientists and engineers is best characterized as having solutions—or mathematical techniques to obtain solutions—in their pockets, and going around looking for problems to fit the solutions or techniques. Now that is not at all the way to solve a problem: a much better way is to start with the problem, and then look for a solution or technique to fit the problem. We should reason from the problem toward mathematics, not from mathematics toward the problem—for problem solvers seek solutions, not merely more mathematics. How much of the compumetric literature on modeling and simulation starts—and for that matter, ends—with mathematics rather than measurement?

Some of the aspects which we should consider are now briefly discussed, from the more general to the more specific (see Figure 1). The discussion might be viewed as a partial characterization of solving problems with mathematics—including its do's and its don'ts. Users should constantly be aware of not only the strengths of mathematics, but also the weaknesses of those who practice it.

In management, science and technology—as well as in mathematics, there are three sequential stages in the search for knowledge. First we explore the unknown, such as go fishing . . . then we try to exploit or utilize what exploration has yielded, such as use the caught fish for reasonable purposes . . . finally we try to explain exploration and exploitation themselves or what they have accomplished, such as describe fishing and fish use or their accomplishments. Exploration seeks the descriptive or indicative, exploitation seeks the predictable or promising, and explanation seeks the provable or conclusive. Each stage has its own philosophy and psychology—which are generally productive for it, but usually unproductive or counterproductive for the others. A successful explorer, a successful merchant and a successful historian essentially accomplish their objectives in different ways—the rules of thumb which generally work for one, usually do not work for the others. It is critical—if not crucial—that we know which stage we are in, and that we act accordingly. Measurement of computer systems is somewhere between exploration and exploitation, but far from explanation.

When resources—money, people or time—are limited, we should first distinguish that which is of primary importance from that which is of secondary importance. We should then expend the available resources on solving the primary portion of a problem, with resources being expended on solving the secondary portion only as they become available. However, we usually commit our scarce resources equally between solving the primary and secondary portions of the problem—to an equally unsatisfactory degree, of whose computer system is this not reminiscent?

The very cornerstone of mathematics is perfection: never settle for an approximate solution when an exact one is possible. A much superior guideline is to never seek an exact solution when an approximate one of sufficient quality is possible—the watchword is K(eep) I(t) S(imple) S(tupid). Or as an old Russian proverb warns, "The enemy of the good is the better." The solution to any problem is only worth so much agony—suffer that much and then stop. Since that has not been the tradition in computer hardware and software development, one wonders what will happen in computer hardware and software measurement.

Analysis is concerned with fragmenting a whole into its component parts—essentially by emphasizing differences and secondary details, while synthesis is concerned with combining constituent parts into a whole—essentially by emphasizing similarities and primary essentials. Analysis frequently leads to refinement: synthesis frequently leads to expansion. Through both education and experience, most of us are analytic whizzes and synthetic flops. Analyzing computer system measurements is one thing, but synthesizing system improvement from this analysis is quite another thing.

Related to this is the distinction between specialty and generality. A specialist knows a lot about a little—possessing depth rather than breadth, but a generalist knows a little about a lot—possessing breadth rather than depth. The major difficulty in being a meaningful generalist is in becoming sufficiently deep to be productive—without becoming deep enough to be preoccupied—in several significant disciplinary areas. There are few of us who can envision the entire problem picture, yet fully appreciate the style of, and technique required by, many of its parts. Those who are computer hardware or software specialists are occasionally knowledgeable about measurement, but are seldom knowledgeable about data modeling or analysis.

Kelley is reputed to have said that effective communication of mathematics involves both precise statement and "vaguing it up." We might well add that effective performance of mathematics itself involves both precise interpretation and vaguing it up. Although the power of mathematics lies in the precision of its language and operations, the usefulness of mathematics lies in our ability to translate that language and adapt those operations to users of mathematics and to their points of view—focusing upon similarities and primary essentials, while obscuring differences and secondary details. But modelers insist upon projecting the image that mathematics and modeling constitute a mystical religion which is led by ordained high priests, who communicate only among themselves and with the gods of mathematics and modeling. They behave as if they are either incapable of, or uninterested in, describing concepts and techniques in easily understood and meaningful terms.

Being effective is obtaining a good solution to the actual problem, and being efficient is obtaining whatever solution to whichever problem in a timely and inexpensive manner. Effectiveness is more externally oriented toward pre-mathematics and post-mathematics, while efficiency is more internally oriented toward mathematics. If we cannot be both effective and efficient, it is far better to be inefficiently effective, than it is to be efficiently ineffective. Measurement of computer systems must relinquish some of its efficiency orientation in favor of some effectiveness orientation, as is advocated in Gruenberger[6] and by the author.[3]

Similarly, accuracy means closeness and is more involved with pre-mathematics and post-mathematics, but precision means repeatability—whether being close or not—and is more involved with mathematics. We are usually lured away from accuracy by precision, just as we are usually lured away from effectiveness by efficiency. To borrow the phrasing from above, being imprecisely accurate is greatly preferable to being precisely inaccurate. How often are inappropriate computer system characteristics measured with extreme precision?

There is an inherent relationship among the manner in which number should be associated with the problem's structure during pre-mathematics, the manner in which number should be utilized through the problem's model within mathematics, and the manner in which number should be interpreted from the problem's solution during post-mathematics. A given introduction of number supports only an analysis and its solution which are "weaker" than a determined maximum, while a certain implementation of an analysis and its solution is supported only by an introduction of number which is "stronger" than a determined minimum. Stevens[7] describes four scales of number association: nominal, ordinal, interval and ratio. Nominal association, such as player numbers on a team, only distinguishes among entities . . . ordinal association, such as house numbers on a street, merely assigns an order among entities . . . interval association, such as temperature in degrees Fahrenheit, just assigns a meaning to differences between entities . . . ratio association, such as temperature in degrees Rankine, also assigns a meaning to ratios of entities. Number utilization and interpretation should not be blind to number association—which, in turn, should not be blind to desired number utilization and interpretation. Compumetricians need to pay serious attention to the inherent relationship between measurement and mathematics.

Just for fun, why not compile your own list of aspects which should be—but all too often, unfortunately are not—considered in solving problems with mathematics in general or data modeling and analysis in particular?

COMPLETE SYSTEM DESIGN AND ANALYSIS

Experimentation regarding many complex and important systems is impossible during their design, and difficult or expensive during their analysis. For such systems, a mathematical solution for output in terms of input usually does not exist, and computer simulation may be effectively employed as a substitute for experimentation during design and as a complement to it during analysis.

The system and the effects of various factors upon it may be simulated when a model of the system or process is translated into a simulation computer program. Accuracy and precision of the simulation increase, as the accuracy and precision of the model increase.

A general approach to utilization of both experimental and computer simulation data in system design and analysis—called complete system design and analysis—is discussed by the author,[8] and the premathematical portion—quantitative system design and analysis—is illustrated by the author.[5] Complete system design and analysis are summarized by Figures 2-17.

The basic configuration of complete system design and analysis is that of a double diamond. Its outer portion (AB, BC, CD, DA, AE and CF) contains those stages which are not data based, and its inner portion (EB, BF, FD, DE and EF) contains those stages which are data based.

The model and simulation computer program are developed and validated by means of stages which comprise the upper part of the double diamond (AB, BC, CF, FE, EA, EB and FB). Analysis of data, and design of experimental and simulation trials to opti-

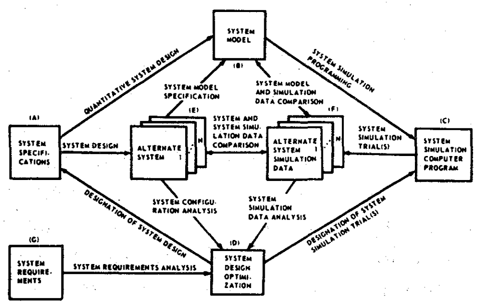
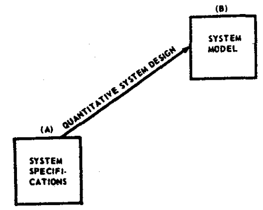

SYSTEM MODEL
(B)
QUANTITATIVE SYSTEM DESIGN
SYSTEM SIMULATION PROGRAMMING
SYSTEM MODEL SPECIFICATION
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(E)
(F)
(A)
SYSTEM SPECIFI-CATIONS
SYSTEM DESIGN
ALTERNATE SYSTEM 1
N
SYSTEM AND SYSTEM SIMU-LATION DATA COMPARISON
ALTERNATE SYSTEM SIMULATION DATA 1
N
SYSTEM SIMULATION TRIAL(S)
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGNATION OF SYSTEM DESIGN
SYSTEM CONFIGU-RATION ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
DESIGNATION OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE-MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM DESIGN OPTIMI-ZATION

Figure 2—Complete system design: Framework for utilization of computer simulation in design of a complex system

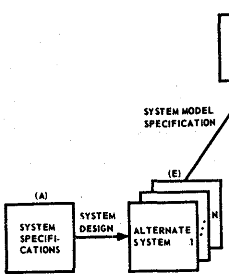


- QUANTITATIVE SYSTEM DESIGN DEVELOPS A SYSTEM MODEL (MATHEMATICAL REPRESENTATION OF THE SYSTEM), WHICH IS SUFFICIENTLY GENERAL TO COVER EACH ALTERNATE SYSTEM, BY TRANSFORMING QUALITATIVE SYSTEM SPECIFICATION ELEMENTS INTO NUMERICAL FORM, AND CONSTRUCTING A MODEL FOR RELATIONSHIPS AMONG SYSTEM SPECIFICATION COMPONENT PARTS.

- TRANSFORMATION OF QUALITATIVE SYSTEM SPECIFICATION ELEMENTS INTO NUMERICAL FORM IS ACCOMPLISHED BY ARRANGING THE ELEMENTS INTO AN INFORMATIVE DETAILED (LOCAL) STRUCTURE, AND THEN ASSOCIATING A MEANINGFUL NUMBER WITH EACH ELEMENT.

- CONSTRUCTION OF A MODEL FOR RELATIONSHIPS AMONG SYSTEM SPECIFICATION COMPONENT PARTS IS ACCOMPLISHED BY ARRANGING THE COMPONENT PARTS INTO AN INFORMATIVE GENERAL (GLOBAL) STRUCTURE, AND THEN SPECIFYING THE GENERAL FORM OF MEANINGFUL RELATIONSHIPS AMONG COMPONENT PARTS.

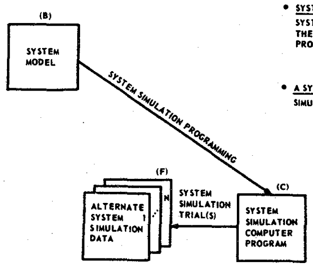
Figure 3—Quantitative system design



- SYSTEM DESIGN PRODUCES ALTERNATE SYSTEMS 1, ..., N, WHICH IMPLEMENT SYSTEM SPECIFICATIONS.

- SYSTEM MODEL SPECIFICATION PRODUCES, FOR EACH ALTERNATE SYSTEM, VALUES OF UNSPECIFIED CONSTANTS IN THE GENERAL FORM OF RELATIONSHIPS IN THE SYSTEM MODEL; AND A PRELIMINARY, BUT INSUFFICIENT, EVALUATION AND VALIDATION (POSITIVE CHECK) OF THE SYSTEM'S REPRESENTATION BY THE SYSTEM MODEL.

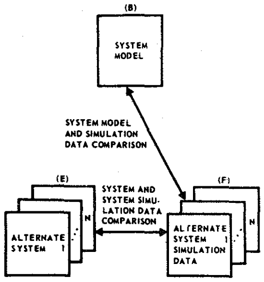
Figure 4—System design and model specification


(B)
SYSTEM MODEL
SYSTEM SIMULATION PROGRAMMING
(F)
ALTERNATE SYSTEM 1 SIMULATION DATA
N
SYSTEM SIMULATION TRIAL(S)
(C)
SYSTEM SIMULATION COMPUTER PROGRAM

- SYSTEM SIMULATION PROGRAMMING TRANSLATES THE SYSTEM MODEL (MATHEMATICAL REPRESENTATION OF THE SYSTEM) INTO A SYSTEM SIMULATION COMPUTER PROGRAM (COMPUTER REPRESENTATION OF THE SYSTEM).

- A SYSTEM SIMULATION TRIAL PRODUCES SYSTEM SIMULATION DATA FOR EACH ALTERNATE SYSTEM.

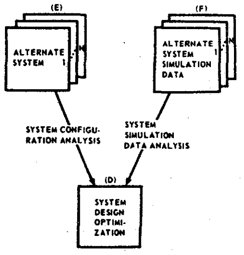
Figure 5—System simulation programming and trial(s)


(B)
SYSTEM MODEL
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(E)
ALTERNATE SYSTEM 1
N
SYSTEM AND SYSTEM SIMU-LATION DATA COMPARISON
(F)
ALTERNATE SYSTEM 1 SIMULATION DATA
N

- SYSTEM MODEL AND SYSTEM SIMULATION DATA COMPARISON PROVIDES, FOR EACH ALTERNATE SYSTEM, AN EVALUATION AND VALIDATION (POSITIVE CHECK) OF THE SYSTEM MODEL'S REPRESENTATION BY THE SYSTEM SIMULATION COMPUTER PROGRAM.

- SYSTEM AND SYSTEM SIMULATION DATA CCMPARISON PROVIDES, FOR ƐACH ALTERNATE SYSTEM, AN EVALUATION AND VAL:DATION OF THE SYSTEM'S REPRESENTATION BY THE COMBINATION OF SYSTEM MODEL AND SIMULATION COMPUTER PROGRAM, AND AN INDIRECT EVALUATION AND VALIDATION OF THE SYSTEM'S REPRESENTATION BY THE SYSTEM MODEL, GIVEN THAT THE MODEL'S REPRESENTATION BY THE SIMULATION COMPUTER PROGRAM HAS BEEN VALIDATED.

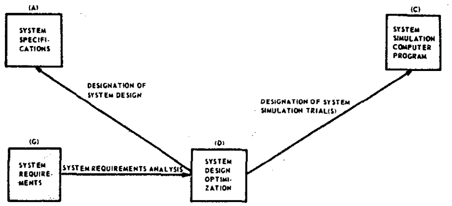
Figure 6—System model, system simulation data, and system comparison


(E)
ALTERNATE SYSTEM 1
N
(F)
ALTERNATE SYSTEM 1 SIMULATION DATA
N
SYSTEM CONFIGU-RATION ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
(D)
SYSTEM DESIGN OPTIMI-ZATION

- SYSTEM CONFIGURATION AND SIMULATION DATA ANALYSIS CHARACTERIZE AND EVALUATE EACH ALTERNATE SYSTEM, AND SUGGEST MODIFICATION OF THE SYSTEM DESIGN FOR SYSTEM DESIGN OPTIMIZATION (ACHIEVEMENT OF SUFFICIENT IMPROVEMENT).

- EVALUATION OF EACH ALTERNATE SYSTEM IS IN TERMS OF SYSTEM CRITERIA AND CONSTRAINTS (FOR EXAMPLE, THOSE OF COST OR EFFECTIVENESS).

Figure 7—System configuration and simulation data analysis


(A)
SYSTEM SPECIFI-CATIONS
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGNATION OF SYSTEM DESIGN
DESIGNATION OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE-MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM DESIGN OPTIMI-ZATION

- SYSTEM DESIGN OPTIMIZATION MODIFIES THE SYSTEM DESIGN, AND APPLIES THE APPROPRIATE PORTIONS OF COMPLETE SYSTEM DESIGN TO THE MODIFIED DESIGN - IN AN ITERATIVE MANNER, UNTIL THE ACHIEVEMENT OF SUFFICIENT IMPROVEMENT.

- DESIGNATION OF SYSTEM DESIGN AND SIMULATION TRIAL(S) AIDS THE IMPLEMENTATION OF SYSTEM DESIGN OPTIMIZATION

- SYSTEM REQUIREMENTS ANALYSIS PROVIDES A BASIS FOR SYSTEM DESIGN OPTIMIZATION, AND DESIGNATION OF SYSTEM DESIGN AND SIMULATION TRIAL(S).

Figure 8—System design optimization, designation of system design and simulation trial(s), and system requirements analysis

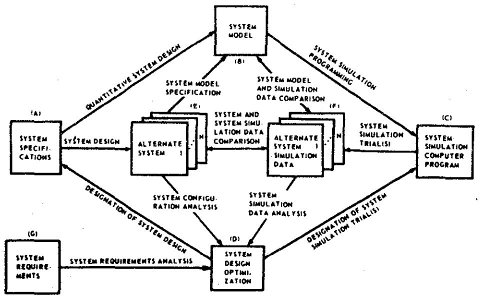

SYSTEM MODEL
(B)
QUANTITATIVE SYSTEM DESIGN
SYSTEM SIMULATION PROGRAMMING
SYSTEM MODEL SPECIFICATION
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(A)
SYSTEM SPECIFI-CATIONS
SYSTEM DESIGN
(E)
ALTERNATE SYSTEM 1
N
SYSTEM AND SYSTEM SIMU-LATION DATA COMPARISON
(F)
ALTERNATE SYSTEM 1 SIMULATION DATA
N
SYSTEM SIMULATION TRIAL(S)
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGNATION OF SYSTEM DESIGN
SYSTEM CONFIGU-RATION ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
DESIGNATION OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE-MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM DESIGN OPTIMI-ZATION

Figure 9—Complete system design: Framework for utilizationof computer simulation in design of a complex system

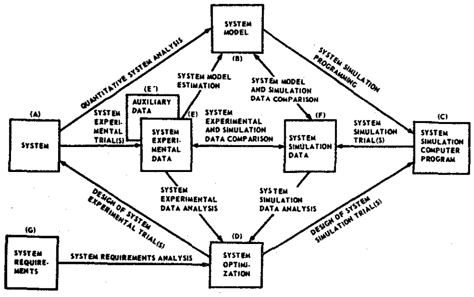
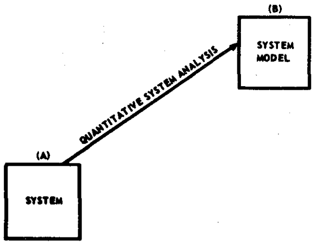
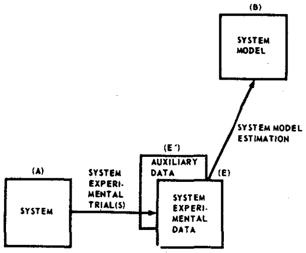

SYSTEM MODEL
(B)
QUANTITATIVE SYSTEM ANALYSIS
SYSTEM SIMULATION PROGRAMMING
SYSTEM MODEL ESTIMATION
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(E')
AUXILIARY DATA
(A)
SYSTEM
SYSTEM EXPERI-MENTAL TRIAL(S)
(E)
SYSTEM EXPERI-MENTAL DATA
SYSTEM EXPERIMENTAL AND SIMULATION DATA COMPARISON
(F)
SYSTEM SIMULATION DATA
SYSTEM SIMULATION TRIAL(S)
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGN OF SYSTEM EXPERIMENTAL TRIAL(S)
SYSTEM EXPERIMENTAL DATA ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
DESIGN OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE-MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM OPTIMI-ZATION

Figure 10—Complete system analysis: Framework for utilization of computer simulation in analysis and optimization of a complex system



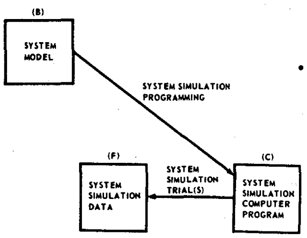
- QUANTITATIVE SYSTEM ANALYSIS DEVELOPS A SYSTEM MODEL (MATHEMATICAL REPRESENTATION OF THE SYSTEM) BY TRANSFORMING QUALITATIVE SYSTEM ELEMENTS INTO NUMERICAL FORM, AND CONSTRUCTING A MODEL FOR RELATIONSHIPS AMONG SYSTEM COMPONENT PARTS.
- TRANSFORMATION OF QUALITATIVE SYSTEM ELEMENTS INTO NUMERICAL FORM IS ACCOMPLISHED BY ARRANGING THE ELEMENTS INTO AN INFORMATIVE DETAILED (LOCAL) STRUCTURE, AND THEN ASSOCIATING A MEANINGFUL NUMBER WITH EACH ELEMENT.
- CONSTRUCTION OF A MODEL FOR RELATIONSHIPS AMONG SYSTEM COMPONENT PARTS IS ACCOMPLISHED BY ARRANGING THE COMPONENT PARTS INTO AN INFORMATIVE GENERAL (GLOBAL) STRUCTURE, AND THEN SPECIFYING THE GENERAL FORM OF MEANINGFUL RELATIONSHIPS AMONG COMPONENT PARTS.

Figure 11—Quantitative system analysis



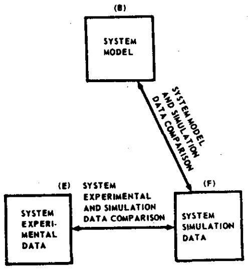
- A SYSTEM EXPERIMENTAL TRIAL PRODUCES SYSTEM EXPERIMENTAL DATA.
- SYSTEM MODEL ESTIMATION PRODUCES ESTIMATES OF UNSPECIFIED CONSTANTS IN THE GENERAL FORM OF RELATIONSHIPS IN THE SYSTEM MODEL; AND A PRELIMINARY, BUT INSUFFICIENT, EVALUATION AND VALIDATION (POSITIVE CHECK) OF THE SYSTEM'S REPRESENTATION BY THE SYSTEM MODEL.

Figure 12—System experimental trial(s) and model estimation


(B)
SYSTEM MODEL
SYSTEM SIMULATION PROGRAMMING
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
SYSTEM SIMULATION TRIAL(S)
(F)
SYSTEM SIMULATION DATA

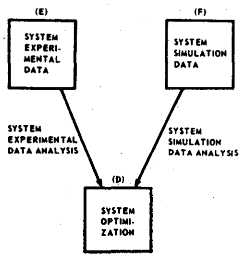
- SYSTEM SIMULATION PROGRAMMING TRANSLATES THE SYSTEM MODEL (MATHEMATICAL REPRESENTATION OF THE SYSTEM) INTO A SYSTEM SIMULATION COMPUTER PROGRAM (COMPUTER REPRESENTATION OF THE SYSTEM).
- A SYSTEM SIMULATION TRIAL PRODUCES SYSTEM SIMULATION DATA.

Figure 13—System simulation programming and trial(s)


(B)
SYSTEM MODEL
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(E)
SYSTEM EXPERI-MENTAL DATA
SYSTEM EXPERIMENTAL AND SIMULATION DATA COMPARISON
(F)
SYSTEM SIMULATION DATA

- SYSTEM MODEL AND SIMULATION DATA COMPARISON PROVIDES AN EVALUATION AND VALIDATION (POSITIVE CHECK) OF THE SYSTEM MODEL'S REPRESENTATION BY THE SYSTEM SIMULATION COMPUTER PROGRAM.
- SYSTEM EXPERIMENTAL DATA AND SIMULATION DATA COMPARISON PROVIDES AN EVALUATION AND VALIDATION OF THE SYSTEM'S REPRESENTATION BY THE COMBINATION OF SYSTEM MODEL AND SIMULATION COMPUTER PROGRAM; AND AN INDIRECT EVALUATION AND VALIDATION OF THE SYSTEM'S REPRESENTATION BY THE SYSTEM MODEL, GIVEN THAT THE MODEL'S REPRESENTATION BY THE SIMULATION COMPUTER PROGRAM HAS BEEN VALIDATED.

Figure 14—System model, simulation data, and experimental data comparison


(E)
SYSTEM EXPERI-MENTAL DATA
(F)
SYSTEM SIMULATION DATA
SYSTEM EXPERIMENTAL DATA ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
(D)
SYSTEM OPTIMI-ZATION

- SYSTEM EXPERIMENTAL AND SIMULATION DATA ANALYSIS CHARACTERIZE AND EVALUATE THE SYSTEM, AND SUGGEST MODIFICATION OF THE SYSTEM FOR SYSTEM OPTIMIZATION (ACHIEVEMENT OF SUFFICIENT IMPROVEMENT).
- EVALUATION OF THE SYSTEM IS IN TERMS OF SYSTEM CRITERIA AND CONSTRAINTS (FOR EXAMPLE, THOSE OF COST OR EFFECTIVENESS).

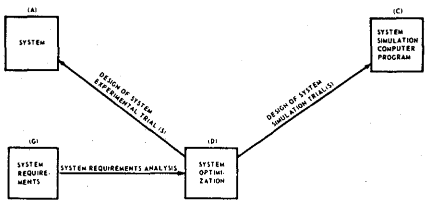
Figure 15—System experimental and simulation data analysis

mize the system, are performed by those stages which comprise the lower part of the double diamond (AD, ED, FD and CD).

Development of the model and design, and performance and analysis of experimental trials are accomplished by those stages on the left side (AB, BE, ED, DA and AE). Finally, the right side (BC, CD, DF, FB, CF and FE) contains those stages concerned with developing and validating the simulation computer program, and with designing, performing and analyzing simulation trials.

The inherent symmetry and simplicity of the double diamond make it a very meaningful and suggestive way in which to view complete system design and analysis. In complete system design (Figure 2) and analysis (Figure 10):

- Modeling is applicable to quantitative system design (AB) and to quantitative system analysis (AB).
- Statistical estimation in general and regression analysis in particular are applicable to system model specification (EB) and to system model estimation (EB).
- Statistical testing in general and analysis of variance in particular are applicable to system model and simulation data comparison (BF), to


(A)
SYSTEM
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGN OF SYSTEM EXPERIMENTAL TRIAL(S)
DESIGN OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE-MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM OPTIMI-ZATION

- SYSTEM OPTIMIZATION MODIFIES THE SYSTEM, AND APPLIES THE APPROPRIATE PORTIONS OF COMPLETE SYSTEM ANALYSIS TO THE MODIFIED SYSTEM - IN AN ITERATIVE MANNER, UNTIL THE ACHIEVEMENT OF SUFFICIENT SYSTEM IMPROVEMENT
- DESIGN OF SYSTEM EXPERIMENTAL AND SIMULATION TRIALS AIDS THE IMPLEMENTATION OF SYSTEM OPTIMIZATION
- SYSTEM REQUIREMENTS ANALYSIS PROVIDES A BASIS FOR SYSTEM OPTIMIZATION,AND DESIGN OF SYSTEM EXPERIMENTAL AND SIMULATION TRIALS

Figure 16—System optimization, design of system experimental and simulation trials, and system requirements analysis

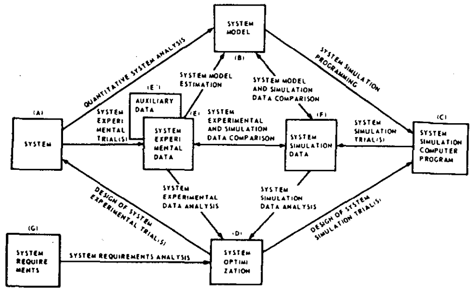

SYSTEM MODEL
(B)
QUANTITATIVE SYSTEM ANALYSIS
SYSTEM SIMULATION PROGRAMMING
SYSTEM MODEL ESTIMATION
SYSTEM MODEL AND SIMULATION DATA COMPARISON
(E')
AUXILIARY DATA
(A)
SYSTEM
SYSTEM EXPERI MENTAL TRIAL(S)
(E)
SYSTEM EXPERI MENTAL DATA
SYSTEM EXPERIMENTAL AND SIMULATION DATA COMPARISON
(F)
SYSTEM SIMULATION DATA
SYSTEM SIMULATION TRIAL(S)
(C)
SYSTEM SIMULATION COMPUTER PROGRAM
DESIGN OF SYSTEM EXPERIMENTAL TRIAL(S)
SYSTEM EXPERIMENTAL DATA ANALYSIS
SYSTEM SIMULATION DATA ANALYSIS
DESIGN OF SYSTEM SIMULATION TRIAL(S)
(G)
SYSTEM REQUIRE MENTS
SYSTEM REQUIREMENTS ANALYSIS
(D)
SYSTEM OPTIMI ZATION

Figure 17—Complete system analysis: Framework for utilization of computer simulation in analysis and optimization of a complex system

system and system simulation data comparison (EF), and to system experimental and simulation data comparison (EF).

- Statistical estimation and testing in general, and regression analysis and analysis of variance in particular, are applicable to system experimental data analysis (ED) and to system simulation data analysis (FD).
- Statistical design is applicable to design of experimental trial(s) (DA), to designation of system simulation trial(s) (DC), and to design of system simulation trial(s) (DC).

ACKNOWLEDGMENTS

The critical review of this paper and constructive suggestions for its improvement by Barry Boehm, Richard Hamming and Stephen Kimbleton are gratefully acknowledged.

REFERENCES

1 R HAAVIND Editor
New Uses for computers in business:Marketing finance administration scientific decision making and planning
Computer Decisions Vol 4 No 1 1972 pages 17-40

2 A F GOODMAN Editor
Computer science and statistics: Fourth annual symposium on the interface—An interpretive summary
Western Periodicals Company 1971

3 A F GOODMAN
Measurement of computer systems—An introduction
Proceedings of 1972 Fall Joint Computer Conference AFIPS Press 1972

4 T E BELL B W BOEHM R A WATSON
Framework and initial phases for computer performance improvement
Proceedings of 1972 Fall Joint Computer Conference AFIPS Press 1972

5 A F GOODMAN
Flow of scientific and technical information: The results of a recent major investigation
Western Division Paper 4516 McDonnell Douglas Astronautics Company 1967 and Revised 1970 (Available from the National Technical Information Service as AD 657 558)

6 F GRUENBERGER Editor
Effective versus efficient computing
Publisher to be selected

7 S S STEVENS
Measurement statistics and the schemapiric view
Science Vol 161 No 40 1968 pages 849-856

8 A F GOODMAN L GAINEN C O BEUM JR
Complete system analysis: Quantitative system analysis computer simulation and system optimization
Western Division Paper 4431 McDonnel Douglas Astronautics Company 1967 and Revised 1969 (Available from the National Technical Information Service as N69-16331)

From PLANNER to CONNIVER—A genetic approach

by GERALD JAY SUSSMAN and DREW VINCENT McDERMOTT

Massachusetts Institute of Technology
Cambridge, Massachusetts

THE PROBLEM WITH PLANNER

A higher level language derives its great power from the fact that it tends to impose structure on the problem solving behavior of the user. Besides providing a library of useful subroutines with a uniform calling sequence, the author of a higher level language imposes his theory of problem solving on the user. By choosing what primitive data structures, control structures, and operators he presents, he makes the implementation of some algorithms more difficult than others, thus discouraging some techniques and encouraging others. So, to be good, a higher level language must not only simplify the job of programming, by providing features which package programming structures commonly found in the domain for which the language was designed, it must also do its best to discourage the use of structures which lead to bad algorithms.

PLANNER[1] is the language designed by Carl Hewitt of the MIT Artificial Intelligence Laboratory. (A subset of PLANNER was rather haphazardly implemented by G. J. Sussman, T. Winograd and E. Charniak. We call this operational system MICRO-PLANNER[2]). PLANNER incorporates three basic ideas; automatic backtrack control structure, pattern-directed data-base search, and pattern-directed invocation of procedures. Basically, backtracking is a way of making tentative decisions which can be taken back if they don't pan out. The pattern-directed data base search allows the user to ask for the data items called *assertions* which match a given pattern, and is intimately linked via the GOAL function to pattern-directed procedure invocation, which gives the user the ability to say "Find and run a program whose declared purpose matches this pattern." This type of program, called a *theorem,* is expected to instantiate the pattern (*succeed*), and thus simulate an assertion. In fact, it simulates a whole class of them, since failures back into any such theorem cause it to make different choices and succeed with different instances.

How these mechanisms are related can best be illustrated by an example. The statement (GOAL (?A IN ?B)) is expected to assign the question-marked variables that do not have values already, or fail if it can't, causing a backup to the last decision point in the program. GOAL instantiates its pattern by matching it against successive assertions, like (BLOCK2 IN BOX1). If it finds none, or enough failures propagate back to the GOAL to use up those it has found, it calls theorems with matching patterns, such as:

```
(CONSEQUENT (X Y Z) (?X IN ?Y)
         (GOAL (?X IN ?Z))
         (GOAL (?Z IN ?Y))  )
```

which expresses one facet of the notion that IN is transitive. A PLANNER program executing (GOAL (BLOCK2 IN ?B)) first checks to see if it "knows" the answer, and if so sets B to it. If not, it binds X to BLOCK2, links Y and B, enters the theorem, and looks for a Z containing BLOCK2 and contained in some Y. Its net effect is to assign a value to B.

If a failure propagates back into the theorem, it finds another Y containing Z, and hence generates another B; enough failures to use up those Y's drive it to find another Z; and a few more will make it and the original GOAL fail themselves. Backtrack control structure is the heart of this apparatus.

Automatic backtracking is implemented as follows: A PLANNER program, as it runs, grows a *chronological* stack of *failpoints* each of which corresponds to either a side effect or a decision point (a place where a choice is made between several alternative possibilities). A failpoint carries with it all information necessary to reconstruct the state of the running process at the time the failpoint was made. It may logically be considered to be a snapshot of that process (though it really saves much less than a copy). At some time, the process may decide to *fail,* perhaps because some decision made at a previous decision point led the process into a blocked state from which there are no viable alternatives. The failure then pops off the latest failpoint on the chronological stack. If this failpoint was a side effect, then it is undone, and the process continues failing. If this failpoint was a decision point, and there is another alternative, execution proceeds from that failpoint with the next choice taken. If there are none the failure continues to propagate. In these terms, GOAL finds exactly one assertion or theorem each time it is reached, but sets a failpoint which regains control if a failure should occur later.

For some time we have been studying PLANNER and the uses to which it has been put, hoping to learn just what modifications would be desirable to the user community. These investigations have led us to decide that this basic control structure of PLANNER is wrong, though its successes indicate that it contains many powerful (and seductive) ideas. This investigation has led to the design and implementation of a new, and hopefully cleaner language, CONNIVER,[3] built largely on the good ideas found in PLANNER.

Here is our thesis: *automatic* backtracking, which occupies a place in PLANNER analogous to that of recursion in LISP,[4] is the wrong structure for the domain for which PLANNER was intended, that is, Artificial Intelligence. We argue that:

1. Programs which use automatic backtracking are often the *worst* algorithms for solving a problem.
2. The most common use of backtracking can almost always be replaced by a purely recursive structure which is not only more efficient but also clearer, both semantically and syntactically.
3. Superficial analysis of problems and poor programming practice are encouraged by the ubiquity of automatic backtracking and by the illusion of power that a function like GOAL gives the user merely by brute force use of invisible failure-driven loops.
4. The attempt to fix these defects by the introduction of "failure messages" (to be explained) is unnatural and ineffective.

Thus we contend that the problem with PLANNER is automatic backtrack control structure. We must stress, however, that PLANNER has introduced many valuable constructs into our way of thinking, the most important of which is pattern-directed search of a hierarchical data base.

Note also that we are not contending that good programs cannot be implemented in PLANNER; that would be absurd. We are only claiming that PLANNER gets in the user's way when he tries to embody certain straightforward concepts in his programs. Nor are we making the weak point that PLANNER occasionally lures foolish programmers into inefficiency. One could try to make this criticism of LISP by pointing out, for example, how it tempts one to write an exponentially exploding, doubly recursive algorithm for computing the *nth* element in the Fibonacci sequence:

```
(DEFUN FIB (N)
    (COND ( (= O N) 1)
          ( (= 1 N) 1)
          (T (+ (FIB (− N 1) )
                (FIB (− N 2) ) ) )  ) )
```

The language has led us astray here, since it discourages writing the iterative algorithm, but this is no condemnation of LISP; the mechanism or recursive control structure, although the wrong one to use in this pathological case, is often both the most natural and the most efficient control structure for the problems of symbolic manipulation that are typical of LISP applications. PLANNER, however, almost forces inefficiency in exactly the applications for which it was designed.

We now consider our points in detail:

1. All will readily admit that a perfectly clever program would do no backtracking; it would know where it was going at each step and never need to undo a bad decision. Good programs that know the structure of their problem domains (such as Moses' SIN[5]) have no need for an ability to thrash about, searching for a good approach (as in SAINT[6]). Pure backtracking (without failure messages) is essentially a mechanism for easily undoing a bad decision in the hope that a better alternative will be found. Thus it is most appropriate to algorithms which make such bad decisions either because of lack of sufficient guiding structure in the problem space or of sufficient knowledge of that structure in the program.

   It is, of course, impossible in practice or in principle to achieve perfect knowledge in most AI application programs. Inevitably, programs will recognize that they have gone seriously agley, and will have to undo part of what they have done. Unfortunately, pure backtracking undoes everything since the last decision, without enquiring as to whether it was the one at fault. Such a program will eventually stumble upon the right path, but its organization makes it hard for it to learn something from an attempt that failed and erased all its side effects. The only attempt at correcting this intrinsic defect of failure in the PLANNER sense is the failure message device, to be discussed under point four.

2. Observation of the MIT vision group's[7] use of MICRO-PLANNER tends to indicate that one of the more important uses of backtracking, in programs which are not searching because they know exactly where they are going, is in information retrieval. Although important, it is curiously quite elementary for such a powerful sounding primitive as GOAL. Vision programs maintain large data bases of information about a visual scene, and often must be able to search out relevant data items from a mass of irrelevancies. For example:

```
(GOAL (?X IS BIG) )
(GOAL (,X IS GREEN) )
(GOAL (,X ON ?Y) )
(GOAL (,Y IS BLUE) )
(stuff X Y)
```

means "do the stuff" on the first objects X and Y such that "the big green X is on the blue Y." If *stuff* doesn't like the first ones found, it can easily fail, hoping to get more if there are any. Note that what is going on here is sequential filtering of the possible assignments of X and Y by pattern-directed search of the data base and theorems. We see that backtracking must be used here because any particular big X chosen on line one may not be green, or may not be on something blue. The stack frame of each goal statement thus maintains a list of the hitherto untried possibilities and if a failure reaches it, it tries the next one and proceeds.

A much more straightforward and revealing approach would be to use ordinary recursive and iterative control structure to filter the possibilities directly. Thus, for example, a LISP function FOR-ALL might be written, such that:

```
(FOR-ALL (?X IS BIG)
        (FOR-ALL (,X IS GREEN)
                (FOR-ALL (,X ON ?Y)
                        (FOR-ALL (,Y IS BLUE)
                                (stuff X Y) ) ) ) )
```

would have the desired effect. Here, FOR-ALL is just a standard LISP function which, upon entry, looks up all of the assertions and theorems matching the pattern given as its first argument (with values substituted for variables which are already assigned). It then assumes the first possibility, assigning variables appropriately, and evaluates its second argument. If the evaluation ever returns, rather than exiting the loop, the first element is removed from the list of possibilities and the process repeats. Notice that by appropriately nesting our loops no backtracking is required in the data retrieval. Here *stuff* is done on each X and Y which satisfies the criteria until *stuff* decides it has had enough, and leaves the nest of FOR-ALL's (with a RETURN, GO, or something similar).

This good nesting of loops has decided advantages. Besides being more efficient than backtracking (a marginal advantage), good nesting makes the scope of the action clear. There is no chance an unexpected failure will propagate back into this code and compute without our explicit programming of this activity. We want to emphasize that this insidious problem is not invented. It is observed by *real* users of MICRO-PLANNER who complain that they cannot understand the behavior of their programs because the flow of control is not explicit in the code. Usually, any choice made in a piece of code doing such filtering eventually fails for the same reason that the first choice did, but backtracking tends to treat all decision points as equally important and tries all possibilities; the only symptom that the program is running amok is that it takes an excessive amount of time to fail. The consequent theorem given suffers from exactly this problem; if called by, e.g., (GOAL (BLOCK2 IN BOX1)), its only possible actions are either to find a Z between BLOCK2 and BOX1, or to fail. Although which Z is found cannot possibly affect subsequent events, a failure back to the theorem will cause it to look up another Z, succeed, and allow its caller to fail again in exactly the same way!

A number of primitives (FINALIZE, STRAIGHTEN, etc.) have been added to PLANNER to give the user some control over backtracking. In some cases these help, in some cases not; what is always a problem is that the structure of a PLANNER program does not reveal what the programmer's intentions are. He must always keep in mind that in effect there is only one gigantic nest of failure-driven loops in any PLANNER process, and every subprogram that might fail is only a tiny piece of it. We think that it is essentially clearer for any looping or nesting structure to be made explicit.

3. As PLANNER is currently organized, it provides a very compact notation in which to encode exhaustive depth-first searches for solutions to problems the programmer understands poorly. Other program organizations, though certainly possible, are more complex and less transparently described. To ameliorate this situation a multiprocessing capability with ports and a hierarchical data base has been added to PLANNER in later versions. This is an important step in the right direction which we develop extensively in CONNIVER. In PLANNER, however, each process is still crippled by the automatic backtracking built into system primitives. The multiprocessing primitives give a breadthwise capability to PLANNER but this is not enough. A clever program does not exhaustively search a problem space in any *a priori* order but is rather guided by what it finds. It is often necessary to examine the assumptions leading to a dead end. If hitting a dead end implies backtracking, undoing the bad assumptions and their consequences, this structure cannot be easily examined. Thus the sophisticated user of PLANNER must continuously avoid backtracking, often programming around system primitives which set failpoints or possibly fail. He must spend time calculating the possible directions from and circumstances under which he could lose control to the automatic control structure. With all that machinery hung on his programs to circumvent the control structure, they look much less understandable; most programmers just can't be troubled.

   We do not ask what is needed in PLANNER to make it more powerful; we ask instead what it is about PLANNER that makes it so difficult to control. Its defaults are chosen throughout so that backtracking must be tediously reckoned with in every case unless the user explicitly prevents it. It is easy to say (as some PLANNER advocates do) that people should write their programs avoiding the temptation to backtrack except when necessary, but it is hard to avoid when the language gives them every opportunity to fall.

4. In order to give the user a modicum of control over the backtrack mechanism, failure messages were incorporated into PLANNER early in its history. The intent is to give a program the ability to fail to a specific point in its history where a failpoint has been set up. Associated with each failure is a message (possibly null), which each failpoint examines when the failure propagates to it, until a failpoint with a matching pattern is found. This does not give the user the ability to perform even the simplest of control functions. The system primitives, like GOAL, which make most failpoints, create them with null message patterns which are transparent to failures with any other message. Thus, it is impossible to fail to a specific goal statement when it is determined that a bad choice was made there. Even worse, it is difficult to control the alternatives that will be chosen if the current choice fails. Suppose, for example, we have a goal which invokes a theorem. This theorem, in probing the search space, discovers something relevant to its further exploration. It would like to edit the list of theorems which are pending in the goal which called it (the alternatives which will be tried if the current theorem fails), deleting some entries and inserting others. It might even wish to sort the list of alternatives according to some general criterion. It has not yet, however, failed, and thus cannot return a failure message. Furthermore, it cannot get at the list of alternatives pending on its failure.

   This is not a fine point of control structure theory; it would be extremely relevant to a PLANNER encoding of a chess program like that of Greenblatt.[8] For example, this program, in an analysis of a move, may discover that it is in danger of being forked. This discovery must change the whole set of criteria by which it judges further alternatives. It must try to make a move which meets the discovered threat, if possible.

   We have been concentrating here on the sloppy interface between failure messages and GOAL, but there is a fundamental difficulty with them that would be encountered even if the user abandoned GOAL altogether. That is, they can't carry enough information. There is no way to fail back with the message: "Process P ran into difficulty T," because process P and its context have been destroyed by the failure. So all the relevant information must be contained in the T part of the message: "Difficulty T." It is clear that including all and only the relevant information is as impossible a job for a subroutine as anticipating the form of every possible failure is for its caller. In fact, the THMESSAGE primitive of MICRO-PLANNER has never been successfully used; it seems to be one

> of those superficially good ideas that prove to be unworkable.
>
> It seems that a failing program has no choice but to make too much information frozen in too global a context, or to flush everything it has discovered and bet all its chips on one message it hopes somebody, somewhere, can figure out. These alternatives do not really alter the blind nature of a failure-driven process, or of several of them. This is probably why they go unused.

At this point it is desirable to abstract our entire discussion away from the particular primitives of PLANNER, and enquire what is gained and what is lost by the use of automatic backtracking. What is gained is clear, and very appealing. In the first place, it provides a mechanism for *generating* alternatives, one at a time, to be used in an effort to accomplish some task. Secondly, it provides a mechanism for eradicating the consequences of accepting an alternative later found to be unstable.

We have already criticized the consequences of this scheme in several ways. Now we shall argue that its basic defect is that it forces the dangerous assumption that the alternatives at each decision point are independent; that (as within all PLANNER primitives) the trial of one of them may produce little or no information which can influence the selection of further alternatives, or the way in which they are run. This is enforced by the eradication of the consequences of a hypothesis when that hypothesis is discarded.

For example, a robot wants to pick up an object, and he has several ways of doing so. In trying the first method, with his right hand, he discovers that the object is hot by seriously burning himself. It is clear that though this method failed he should not go back and try his left hand. Nor should it be necessary for him to have foreseen the difficulty and thus set up a message catcher for burnt hand failure (or for lightning striking); such caution, applied to all possible contingencies, is impossible. The reasonably designed robot will drastically modify his behavior at this point, say be getting a pair of tongs, after screaming.

Notice also that any failure-driven generator (a function that returns a value but sets a fail-point) is constrained to generate alternatives one at a time. If the alternatives are interdependent, surely the best one should be chosen while all or most are in view. In fact, the only reason for *generating* objects rather than just making a list of them is that sometimes the number of possibilities (as, say, the prime numbers) may be infinite, or the cost of generation of the next possibility is much greater or grows much faster than the cost of testing its usefulness. In many cases, however, an explicit list of all or some of the alternatives is what is desired. Of course, even in PLANNER, such a list must exist, snuggled inside some GOAL's failpoints, but there is no natural way to access it.

The PLANNER implementation of pattern-directed procedure invocation reinforces these problems of backtracking. The anonymity of the procedures that may be fetched by pattern-directed call makes it even easier to pretend to have many "independent" methods of solving the same problem, hoping that one of the methods, to be found by failure, will come up with an acceptable solution. Not only does this organization force each method to have to be able to run in complete ignorance of what has been tried so far, or even that other methods exist, but in many cases the "independent" methods will come up with the same unacceptable answer more than once, causing the system to thrash ridiculously. The solution, of course, is to abandon the myth that there could be several independent methods of attacking any interesting problem, and concentrate on techniques for making methods interact reasonably.

This is not to say that the pattern-directed function call does not have its place in the arsenal of problem solving. It is valuable whenever, either due to the infiniteness of the set or the economics of storage vs. computation, a procedure can be used to represent a set of assertions.

There are many excellent ideas in PLANNER. They include the notions of "generator" and "possibilities list." But they have been pushed far beneath the surface, so that the user may think in terms of "goals." While the concept of goal-directedness is certainly as well established as any in our field, it seems clear that naming a primitive function "GOAL" is not enough to capture the essence of the idea. In the next section, we shall concentrate on isolating the best ideas in PLANNER, discarding those that have gotten so many MICRO-PLANNER users in trouble, starting with automatic chronological backtracking.

BUILDING CONNIVER

We have shown in the first section that backtracking is a device of questionable usefulness at the very tasks for which it was designed. It encourages a linkage of the mechanism for generating alternatives with the mechanism that restores the environment after the investigation of each one. Each time, the generation of the next must proceed on the basis of very little information besides the fact that the last failed. We have, in the end, a control structure that almost forces the user to regard all his problem-solving methods as independent.

It seems to be the linkage of these two mechanisms in the GOAL statement that is at fault. As an alternative, imagine that we are not allowed to clean things up upon failure; that everything each goal-directed subroutine does *stays* done. Then, if the speculation it has indulged in is not to have effects on the environment of its caller (the program considering the alternatives), it must have a *local environment* of its own to make changes to. These changes may make its model of the problem conflict with its superior's model, or may simply be hypothetical additions to it. The important point is that a simple *return* to the caller will be sufficient to make the changes invisible.

This concept can be made clear by analogy with the familiar notions of "control environment" (a stack, for example), and "access environment" (where variables are bound; the term is Bobrow and Wegbreit's;[9] in CONNIVER, we generalize the latter to "data base environment," or *context.* Just as LISP 1.5 supports a tree of access environments ("association lists"), so CONNIVER supports a tree of contexts, in which each daughter-context represents a data base incrementally different from her parent.

This tree, it will be made clear, must be grown and maintained in conjunction with a control environment of equal complexity. But the control structure exists only to exploit the data base, so we return to it later.

Conniver contexts contain *items,* which are simple list structures like PLANNER's assertions (without the theorem-proving connotations that surround the latter term). An item such as (SQUARE48 PAWN3) may be added to the current context with

```
(ADD '(SQUARE48 PAWN3) )
```

and taken out with

```
(REMOVE '(SQUARE48 PAWN3) ).
```

The arguments to ADD and REMOVE are *skeletons,* list structures that define items after substitution of the values of their variables. Variables are indicated by ",". Thus, if X = PAWN2, (ADD '(SQUARE49 ,X)) adds the item (SQUARE49 PAWN2) to the current context.

Now, if the presence or absence of an item is to be reflected only in a local data base, that is, be "hypothetical," the data environment must be "pushed down" before doing ADD's and REMOVE's of this sort. Since, in CONNIVER, a context is a data type, and the current context is assigned to the variable CONTEXT, all we need to write is:

```
(PROG "AUX" ((CONTEXT (PUSH-CONTEXT)))
    (ADD '(SQUARE48 PAWN3))
            .
            .
            . )
```

CONNIVER syntax is roughly that of LISP, but a declaration of local variables must be explicitly indicated with the atom "AUX", and each such local must be given an explicit initial value, if it is not to be unassigned, by being declared as "(variable value)" instead of just "variable." This PROG thus *rebinds* CONTEXT to the value returned by the system function PUSH-CONTEXT. The current context has had one more *context-frame* pushed onto it. New changes apply to this frame only. After the body of the PROG has been executed in this "hypothetical" context, the PROG's control frame will be exited. CONTEXT will be *unbound,* restoring its old value, in which the action of the ADD is invisible; in effect, a *data frame* has been exited as well.

From now on, we shall abbreviate a construction such as the above as

```
(ASSUMING (SQUARE48 PAWN3). . .),
```

to emphasize concisely the nature of the computation ". . ." as proceeding in a slightly new environment. However, it should not be forgotten that, since contexts are data structures, they can be returned as values of functions, assigned to global variables, etc., so that in fact the user has available a *tree* of contexts his program has left behind, in the same way that using functional arguments (closures of functions) in LISP creates a tree of variable-value associations.

Items can be retrieved from the current context by means of the CONNIVER primitive FETCH, which finds all items present in the context that match a pattern. For example, if we let the presence of the item (obj1 ON obj2) mean "object obj1 is resting on top of object obj2," we can find all the objects on BOX2 with

```
(FETCH '(?X ON BOX2))
```

Roughly as in PLANNER, the "?" indicates that the variable X is to be assigned a value by matching the pattern (?X ON BOX2) against some item. However, FETCH does not make the assignment. Since backtracking has been exorcised from CONNIVER, it simply returns a *possibilities list* which points to *all* the matching items, rather than hiding them in a failpoint in GOAL, to be handed to us coyly, one per failure. The user can manipulate this list in any way he chooses; one way is with the system function TRY-NEXT, which pops off and returns the first item in the list, and assigns the pattern variables as the possibility directs. For example, if PYRAMID6 is the first object found, (TRY-NEXT (FETCH '(?X ON BOX2))) sets X to PYRAMID6.

A useful "canned-loop" function, which is implemented using FETCH and TRY-NEXT, is FOR-EACH, defined so that

```
    (FOR-EACH pattern. . .)
```

performs the computation ". . ." for each assignment of the pattern's variables corresponding to an item in the data base. Hence,

```
    (FOR-EACH (?X ON TABLE)
        (PRINT X))
```

prints the names of all objects resting on TABLE.

As in PLANNER, we want to be able to include a set of items in the current context on the basis of some procedural criterion instead of their actual presence. In PLANNER, this ability was attained by the use of consequent theorems, which behave as failure-driven assertion-generators when invoked by GOAL statements. Since there is no backtracking in CONNIVER, the analogous CONNIVER structure, the *if-needed method*, must return all of its possibilities in a single bundle. The simplest type of if-needed uses the primitive NOTE to save the current instantiation of its pattern with the values of the variables, which were not assigned when it was entered, but have been computed by the method. When the method is exited, it returns a possibilities list which includes all the NOTEd instances.

For example, if the presence of an item of the form (obj1 SUPPORTS obj2) is to mean "obj1 is helping maintain obj2 in its present position," the following method expresses part of the idea that if an object is on another, it is supported by it:

```
    (IF-NEEDED SUP-ON (!X SUPPORTS !?Y)
            "AUX" (X Y)
        (FOR-EACH (?Y ON ,X) (NOTE) ))
```

The "!" and "!?" prefix characters mark method-pattern variables that are to receive or to avoid receiving a value, respectively, when the method is entered. The "!?"-variables are to be assigned in the body of the method (here, by the TRY-NEXT of a FOR-EACH loop); hence, this method returns a list of all generated item possibilities for Y's supported by a given X.

Methods like SUP-ON are treated by the system, as other data, as present or absent in a given context. If a method matching a FETCH or FOR-EACH pattern is found, a method possibility is put into the possibilities list produced, to be invoked by TRY-NEXT when the possibilities before it are used up. Upon returning, the method replaces itself on the caller's possibility list by the instances it generated. Hence, the presence of SUP-ON in a context represents the presence of the items it can produce on demand.

As an illustration, suppose the items (TABLE SUPPORTS APPLE3), (PLATE1 ON TABLE), and (BOX2 ON TABLE) are present. Then the loop

```
  (FOR-EACH (TABLE SUPPORTS ?OBJ)
      (PRINT OBJ))
```

prints

```
  APPLE3
  PLATE1
  BOX2,
```

with the method SUP-ON simulating the two items for the second two objects.

A more complex method may require the creation of a hypothetical data base:

```
(IF-NEEDED SUP-UN (!X SUPPORTS !?Y)
        "AUX" (X Y)
   (ASSUMING (,X VANISHES)
       (FOR-EACH (PHYSICAL-OBJECT  ?Y)
           (COND ((UNSTABLE Y) (NOTE))))))
```

This method creates a hypothetical context in which X no longer exists, and sees which objects are no longer stably immobile according to the (sophisticated) function UNSTABLE. These are NOTEd, and, as before, *all* objects X supports (by this criterion) are found before SUP-UN returns.

Our concept of generator appears *simpler* than PLANNER's; methods like these dump all their instances into the caller's possibilities list and return, leaving their control environments and any bindings of CONTEXT to be collected as garbage. Even if its caller wants only one new item possibility, such generators give him all of them. The scheme we have described does not allow the caller to influence the order or selection of objects to be generated. If each generation is expensive, as, in SUP-UN, a call to UNSTABLE might very well be, a lot of unnecessary overhead may be incurred if not all members of the set at issue are wanted. If, in fact, this set is infinite, the scheme breaks down completely.

We have returned to our original problem: how can we maintain in existence the control and context structure of a generator while returning from it with only a few of the possibilities it can find? The answer lies in the structure and function of the possibilities list; to invoke a method found in such a list is to *replace* the method by its value, itself a list of possibilities. If this value list contains a *generalized tag* back to the generator's activation, its environment will be preserved and accessible. Not only that, but if TRY-NEXT comes upon such a thing in a possibilities list, it is bound to transfer control to it. Now the method can generate items in finite groups, asking to be reawakened if none of the items satisfies its caller. A new version of SUP-UN that works this way looks like:

```
(IF-NEEDED SUP-UN2 (!X SUPPORTS !?Y)
    "AUX" (X Y)
  (ASSUMING (,X VANISHES)
    (FOR-EACH (PHYSICAL-OBJECT ?Y)
      (COND     ((UNSTABLE Y)
                 (NOTE)
                 (AU-REVOIR))  )))  ).
```

AU-REVOIR causes an immediate return from the method when the first Y is found, but also returns a tag to its own activation. Remember that SUP-UN2 is interacting with a possibilities list (perhaps hidden in a FOR-EACH) at a higher level. The method itself was found there, representing a set of simulated items; now when it returns, it leaves one new supportee, plus an AU-REVOIR tag which similarly represents the set of *remaining* possibilities it knows about. From the point of view of a FOR-EACH loop, this type of possibilities list is equivalent to the previous exhaustive list, but it differs in several crucial ways.

First, the list is as short as the generator wishes to make it, no matter how large the set it can generate if requested. Second, such a list represents a generation-in-progress which is not complete; the calling process that asked for it is in a position to intervene and advise the generator how to proceed. Third, an AU-REVOIR tag can be treated explicitly as a datum representing a parallel problem investigation and associated world-view. A generator's caller can use such a tag to do relative evaluations, close functions, and fiddle with its binding of CONTEXT. Notice, for example, that a program that uses SUP-UN2 has available a pointer to an incompatible environment where the object X no longer exists; a more sophisticated version could use this ability to communicate the context and remaining physical object possibilities to take advice from its caller on how to generate more objects supported by X.

The requirement that there be *generalized tags*, tags that mention whole control environments, makes it necessary that CONNIVER maintain a control tree similar in structure to the context tree it serves. All such still-viable environments form a set of processes cooperating to solve a problem. Some of these are generators, using possibilities lists as communication channels with their callers, but this by no means exhausts the alternative ways of interacting. In particular, CONNIVER's *generalized control structure* makes it easy to put all of failure and backtracking back in if the user wants them, but he has the duty (or privilege) of designing and maintaining control over what he builds.

A couple of points remain to be made. Notice that, although the loops in SUP-ON and SUP-UN are exhaustive and blind, they are *explicit.* The only natural way to write these generators in PLANNER is by use of successive GOAL statements that filter out the bad choices. Although the user may intend a loop like a FOR-EACH, and, locally, a GOAL conglomeration behaves like one, it suffers uncontrollably from the effects of global failures.

Generators do not have to be methods; we have only been pursuing this example because of the PLANNER analogy; it seems much more rational that the support finders be *generator functions* of one argument X, with values corresponding to Y. (See Reference 3.) There are several such points of CONNIVER style on which we have been deliberately misleading in order to make a simple point about how a kind of generation problem ought to be approached. We do not, in fact, condone the use of exhaustive FOR-EACH loops, or the hiding of basic machinery such as FETCH and TRY-NEXT from the user. Each programmer must confront the problems and powers of the primitive routines in order to devise his own higher-order functions in an informed way.

Communication between processes is essential to the effective use of multiprocessing. We have tried to build as many communication devices into TRY-NEXT and generators as possible in hopes that they will be used. It would be very dangerous to try extending the exhaustive searches used in SUP-UN to something as much more complicated as a plausible chess move generator. A clever generator must be able to talk to its caller. Even in SUP-UN, it seems clear that the order and methods of generation of physical objects should depend to some degree on the problem situation, including the use to which the objects are to be put.

We have constructed CONNIVER partly by raising to prominence ideas casually embedded in PLANNER, partly by reformulation of the primitives to give hidden PLANNER constructs to the programmer, and partly by concentrating on what is needed in a programming language as opposed to a theorem-prover. Our major contribution, we think, is the elimination of automatic backtracking upon failure as a mechanism for the generation of alternative approaches to a problem. We have shown how PLANNER makes it difficult to write controllable programs; how, like most theorem-provers, it is committed to loosely guided exhaustive search as a problem-solving method; and how the user must either succumb to the will of the control structure or spend much of his time using primitives (like FINALIZE, STRAIGHTEN, TEMPROG, *etc., ad infinitum*) that save him from it.

ACKNOWLEDGMENTS

We must deeply acknowledge the profound influence of Joel Moses on this paper. Some of the ideas here are directly due to him; others were independently arrived at by him. Most of our ideas were arrived at by *observation* of *real* users of MICRO-PLANNER. We thank especially Carl Hewitt, many of whose structures we used in the design of CONNIVER. Dan Bobrow, Jeff Rulifson, Bob Balzer, Chris Reeve, Marvin Minsky, Seymour Papert, Tom Knight, Terry Winograd, Richard Greenblatt, Donald Eastlake, David McDonald, Jon L. White, and William Gosper provided valuable sounding boards for these ideas.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the office of Naval Research under Contract Number N00014-70-a-0362-0003.

REFERENCES

1 C HEWITT
PLANNER: A language for manipulating models and proving theorems in a robot
MIT AI Memo 168 (rev) 1970
See also D E Walker and L M Norton (eds) Proc IJCAI 1 pp 295-301

2 G J SUSSMAN T WINOGRAD E CHARNIAK
MICRO-PLANNER reference manual
MIT AI Memo 203A

3 D McDERMOTT G J SUSSMAN
The CONNIVER reference manual
MIT AI Memo 259 1972

4 J McCARTHY et al
LISP 1.5 programmer's manual
MIT Press Cambridge 1962

5 J MOSES
Symbolic integration
MIT Cambridge Mass PhD dissertation 1967

6 J SLAGLE
A computer program for solving problems in freshman calculus (SAINT)
MIT Cambridge Mass PhD dissertation 1961
Also in EA Feigenbaum and J Feldman (eds) Computers and Thought McGraw-Hill pp 191-203

7 P H WINSTON
The MIT robot
D Michie and B Meltzer (eds) Machine Intelligence 7 1972

8 R GREENBLATT et al
The Greenblatt chess program
Proc FJCC pp 801-810 1967
Also MIT AI Memo 174 1969

9 D G BOBROW B WEGBREIT
A model and stack implementation of multiple environments
Bolt Beranek and Newman Inc Report No 2334 1972

10 C HEWITT
Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot
MIT Revised PhD dissertation April 1972 AI Technical Report 258

11 C HEWITT
Procedural embedding of knowledge in PLANNER
Proc IJCAI 2 pp 167-182 Sept 1971

The QA4 language applied to robot planning*

by JAN DERKSEN, JOHNS F. RULIFSON, and RICHARD J. WALDINGER

Stanford Research Institute
Menlo Park, California

INTRODUCTION

This paper introduces an implemented version of a problem-solving language called QA4[1,2] (Question Answerer 4) and illustrates the application of that language to some simple robot problems. This application is especially appropriate, because the QA4 language has features that are recognized as useful for problem-solving programs;[3] these features include built-in backtracking, parallel processing, pattern matching, and set manipulation. Expressions are put into a canonical form and stored uniquely, so that they can have property lists. A context mechanism is provided, so that the same expression can be given different properties in different contexts. The QA4 interpreter is implemented in LISP and can interface with LISP programs. The language is especially intended to be useful for research leading to program verification,[4] modification, and synthesis,[5] to semantically oriented theorem proving,[6] and to various forms of robot planning.[7]

DESIGN PHILOSOPHY

QA4 has been designed with a specific problem-solving philosophy, which it subtly encourages its users to adopt, and which is an outgrowth of our experience with its antecedent, QA3.[8] QA3 contained an axiom-based theorem prover, which we attempted to use for general-purpose problem solving. However, all knowledge had to be stored in the declarative form of logical axioms, with no indication as to its use. When a large number of facts were known, the knowledge could not be used effectively. The system became swamped with irrelevant inferences, even when supplied with several sophisticated syntactic strategies.

In contrast, QA4 can store information in an imperative form, as a program. This makes it possible to store strategic advice locally rather than globally: In giving information to the system, we can tell it how that information is to be used. Strategies tend to be semantic rather than syntactic: We are concerned more with what an expression means than with how long it is. QA4 programs are intended to rely on an abundance of know-how rather than a large search in finding a solution. We expect our problem solver to make few poor choices, and we try to give it all the information at our disposal to restrict these choices.

There are many similarities both in detail and in philosophy between QA4 and a language designed at MIT called PLANNER,[7] a subset of which has been implemented and has been very successful for expressing programs to manipulate building blocks. We have adapted some PLANNER features for our own uses, and some features shared by both languages have been arrived at independently. The fact that the same devices have been found useful by different groups of people and for diverse problem domains encourages us to believe that languages of this sort will have appeal for problem solvers in general.

THE ROBOT PROBLEMS

We will now examine the kind of knowledge we expect a robot planner to have, and the class of problems we expect it to be able to solve. We will consider some problems of a type recently approached by the SRI robot.[4] We will then be better able to discuss the application of QA4 to this domain and the merits of the QA4 approach.

* The research reported herein was sponsored by the National Aeronautics and Space Administration under Contract NASW-2086.

We envision a world consisting of several rooms and a corridor, connected by doorways. There are boxes and other objects in some of the rooms, and there are switches that control the lights. The robot can move freely around the floor, can pass between the rooms, can see and recognize the objects, can push all the objects, and can climb up onto the boxes. If the robot is on top of a correctly positioned box, it can switch the light on and off.*

The first problem faced by the robot is to turn on the light in one of the rooms. To solve this problem it must go to one of the boxes, push the box next to the light switch, climb up on the box, and turn the switch.

We supply the problem solver with a *model* or representation of the world, which includes the arrangement of the rooms and the positions of and relationships between the objects. Furthermore, corresponding to each action the robot can take, we supply an *operator*, whose effect is to alter the model to reflect the changes the robot's action makes on the world. Each operator has *preconditions*, requirements that must be satisfied before it is applied.

For example, the *pushto* operator corresponds to the robot's action of pushing a box. It changes the model by changing the location of the robot and the location of the box. Its precondition is that the robot be next to the box before the operator is applied.

The *goal* of the problem is a set of conditions that we want the model to satisfy. For example, in the problem of turning on the light, we require that the light be on when the task is completed. The problem of planning, then, amounts to the problem of finding a sequence of operators that, when applied to the initial world model, will yield a new model that will satisfy the goal conditions. If the robot then executes the corresponding sequence of actions it will, presumably, have solved the problem.

Let us suppose that the problem solver works backwards from its goal in its search for a solution. It finds an operator whose effect is to change the model in such a way that the goal condition is satisfied. However, the preconditions of that operator might not be true in the initial model. These preconditions then become subgoals, and the problem solver seeks out operators whose effect is to make the subgoals true. This process continues until all preconditions of each operator in the solution sequence are true in the model in which that operator is applied, and thus, in particular, the preconditions of the first operator in the plan are true in the initial model.

THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

Let us examine within this framework the complete solution of the problem of turning on a light. We assume that, in the initial model, the light switch, the robot, and at least one box are all in the same room. We assume that the problem solver can apply a set of operators that includes the following: *turnonlight, climbonbox, pushto,* and *goto,* which correspond to the actions necessary to turn on the light. The goal is that the status of the light be ON. The operator *turnonlight* has the effect of making this condition true. However, the preconditions of this operator, that the box be next to the light switch and the robot be on top of the box, are not true in the initial model. These preconditions therefore become new subgoals. For the robot to be on top of the box, it suffices to have applied the *climbonbox* operator. However, this operator has the precondition that the robot be next to the box; this precondition becomes a new subgoal.

Both this subgoal and the unachieved precondition of the *turnonlight* operator, that the box be next to the light switch, are achieved by the *pushto* operator, which can move a box anywhere in the room. However, the *pushto* operator still has the precondition that the robot be next to the box. This new subgoal can be achieved by the *goto* operator, which can move the robot anywhere around the room. The only precondition of the *goto* operator is that the robot be in the same room as its destination, but this condition is satisfied in our initial model, since the robot and the box are assumed to be in the same room. Thus, a solution has been found: the sequence *goto* the box, *pushto* the box next to the light switch, *climbonbox*, and *turnonlight.*

The QA4 solution to the robot problems is a direct translation of the approach of the STRIPS problem-solving system,[7] which uses the above framework. STRIPS is the problem solver that does the planning for the SRI robot. The solution of the first three problems approached by STRIPS was the first exercise for the QA4 language. The operators were encoded in the QA4 language, and the model was expressed as a sequence of QA4 statements. This package of information, with no further supervision or strategy, sufficed for the solution of the three sample problems. Finding the plans amounted to evaluating the goals expressed in the QA4 language. The solutions were found quickly and with no more search than necessary. More signifi-

* Actually, the robot that exists at SRI can neither climb boxes nor turn switches.

cantly, the operator descriptions were written quickly and are concise and fairly readable.

The STRIPS representation

For STRIPS, the model is a set of sentences in first-order logic; the preconditions of an operator are also expressed as a set of first-order sentences. The description of the operator itself is restricted to a rather rigid format: There is a delete list, a set of sentences to be deleted from the old model, and the add list, a set of sentences to be added to the new model. The delete list expresses facts that may have been true before the action is performed but that will not be true after the action has been completed. The add list expresses facts that might not have been true before the action is performed but that will be true afterwards. In STRIPS, the *turnonlight* operator, for instance, is described as follows: Its preconditions are that the robot be on the box and that the box be next to the light switch. It deletes from the model the fact that the light is OFF, and it adds to the model the fact that the status of the light is ON. In STRIPS, the strategy for selecting and forming sequences of operators is embodied in a large LISP program. The applicability of operators and the differences between states are frequently determined by a general-purpose, first-order, theorem prover, and the operators themselves are coded in a special-purpose Markov Algorithm language. In QA4, all these elements of the problem-solving system can be handled within a single formalism. We can use the full power of the QA4 programming language to construct the operator description. To describe the operators we have discussed above, we follow the STRIPS format rather closely. For more complex operators and plans, we may make use of more of the language features, as we shall see below.

The QA4 representation

We will now look at the QA4 program for the *turnonlight* operator;* the reader can thus become familiar with the flavor and some of the features of QA4 without having to read a general description:

```
(LAMBDA (STATUS ←M ON)
  (PROG (DECLARE N)
    (EXISTS (TYPE $M LIGHTSWITCH))
    (EXISTS (TYPE ←N BOX))
    (GOAL DO (NEXTTO $N $M))
    (GOAL DO (ON ROBOT $N))
    ($DELETE (′ (STATUS $M OFF)))
    (ASSERT (STATUS $M ON))
    ($BUILD (′ (:$TURNONLIGHTACTION
      $M))))).
```

First, we summarize the action of this operator on the model: It selects a box and asks that the box be next to the light switch and that the robot be on top of the box. It then turns the light on, and it adds the turning of the switch to the sequence of actions to be executed by the robot.

The reader will note how concise and readable the QA4 representation of operators is. Now we will examine the *turnonlight* operator in more detail, to see what it does and the constructs it uses.

The pattern

The program has a LISP-like appearance, but it is evaluated by a special interpreter. In place of a bound variable list, it has a pattern (STATUS ←M ON). This pattern serves as a relevancy test for application of the function. An operator will be applied only to goals that match its bound variable pattern. This operator will be applied only when something is to be turned on. In STRIPS, the add list serves the same function as the pattern. However, in QA4 the relevancy test is distinct from the changes in the model.

All variables in QA4 have prefixes; the prefix ← of the variable M means that the pattern element ←M will match any expression, and M will then be bound to that expression. The other two pattern elements, STATUS and ON, have no prefixes: They are constants and will match only other instances of themselves.

For the following example we shall assume that we want to turn on LIGHTSWITCH1; our goal, therefore, is (STATUS LIGHTSWITCH1 ON). The pattern of the *turnonlight* operator matches this goal, binding M to LIGHTSWITCH1.

Patterns play many roles in QA4; they may appear on the left side of assignment statements and in data base queries. The ability to have a pattern as the bound variable part of a function gives us a concise notation for naming substructures of complex arguments. It

* The QA4 programs for the other operators, the precise formulation of the light switch problem, and the tracing of the solution of that problem are included in the appendix.

also gives us a flexible alternative to the conventional function-calling mechanism, as we shall see.

Searching the data base

The program must first be sure that the value of M is a light switch. This is one of the preconditions of the operator. The statement (EXISTS (TYPE $M LIGHTSWITCH)) searches for instances of the pattern (TYPE $M LIGHTSWITCH) that have been declared TRUE in the data base.

The $ prefix of the variable M means that $M will match only instances of the value of M; M will never be rebound by this match. Thus, in our example we look only for the expression (TYPE LIGHTSWITCH1 LIGHTSWITCH) in the data base.

Unless otherwise specified, the EXISTS statement also checks that this expression has been declared true. Expressions have values; to declare an expression true, we use the ASSERT statement. This construct sets the value of its argument expression to TRUE. This value is stored in the property list of the expression. In QA4, the model is the set of expressions with value TRUE. For our example, we assume that the user has input (ASSERT (TYPE LIGHTSWITCH1 LIGHTSWITCH)) before attempting the problem. Thus, the fact that LIGHTSWITCH1 is a light switch is included in the model.

The EXISTS statement will cause a failure if no suitable expression is found in the data base. A failure initiates backtracking: Control passes back to the last point at which a choice was made, and another alternative is selected. Much of the power of the QA4 language lies in its implicit backtracking, which relieves the programmer of much of the bookkeeping responsibility.

Choosing a box

The operator uses another EXISTS statement to choose a box to use as a footstool: (EXISTS (TYPE ←N BOX)) searches the data base for an expression of the form (TYPE ←N BOX) whose value is TRUE. That such a box exists is one of the preconditions of the operator. Note that here the variable has prefix ←, so there is a class of expressions the pattern will match, and the variable N will be bound by the matching process. We will assume that (TYPE BOX1 BOX) has been asserted to be true, and that N is bound to BOX1.

If for some reason the operator is unable to use BOX1, a failure will occur. Control will pass back to the EXISTS statement, which will then select another box.

Moving the box

The operator now insists as one of its preconditions that the chosen box be next to the light switch. For this purpose it uses the GOAL construct, a mechanism for activating appropriate functions without calling them by name. To move the box, the operator uses (GOAL DO (NEXTTO $N $M)).

The GOAL first acts as an EXISTS statement: It checks to see whether (NEXTTO $N $M), that is, (NEXTTO BOX1 LIGHTSWITCH1), is in the data base. If BOX1 is already next to the light switch, the goal has already been achieved. However, in general, it will be necessary to move the box by using other operators. In other words, the precondition is established as a subgoal.

Every operator has a bound variable pattern and a "goal class," a user-defined heuristic operator partition. A GOAL statement specifies an expression and a goal class. In these problems there are two goal classes, DO and GO. The operators in the GO class are those that simply move the robot around on the floor: for example, *goto.* The operators that move objects or that cause the robot to leave the floor are in the DO class: *pushto, climbonbox,* and *turnonlight.*

To put an operator in a goal class we input (TO goal-class operator). For instance, for this example we assume that we have input (TO DO CLIMBONBOX).

An operator can only be applied to a goal if it belongs to the goal class specified by the GOAL statement. In our example the goal class is DO. Therefore, the only operators that can be applied are *pushto, climbonbox,* and *turnonlight.*

Each of the operators has a bound variable pattern. To be applied to a goal it is not sufficient that the operator belong to the specified goal class; it is also necessary that the bound variable pattern of the operator match the expression specified by the goal statement. In this case the bound variable pattern of the *pushto* operator, (NEXTTO ←M ←N), matches the goal expression (NEXTTOBOX1 LIGHTSWITCH1), with M bound. Therefore, the *pushto* operator is activated.

We will be somewhat more sketchy about the operation of the *pushto* operator, since our aim is to focus attention on the *turnonlight* operator. The *pushto* operator establishes another subgoal, (NEXTTO ROBOT BOX1), with goal class GO. This goal activates the operator *goto,* which succeeds without establishing any further subgoals.

The GOAL mechanism is powerful because we need not know in advance which functions it will activate; that choice depends on the form of the argument. The relevant operators come forward at the appropriate time.

The *turnonlight* operator requires not only that the box be next to the light switch, but also that the robot be on top of the box, before it can turn on the light. This precondition is described as (GOAL DO (ON ROBOT $N)). Of the operators of the DO class, the only one whose bound variable pattern matches the goal expression is *climbonbox*, with pattern (ON ROBOT ←M), so this operator is applied. The preconditions of this operator, including the requirement that the robot be next to BOX1, are already satisfied in the model. Thus, the operator can report a quick success.

The remaining statements effect the appropriate changes in the model. The statement ($DELETE (′ (STATUS $M OFF)) corresponds to the specification of the delete list in the STRIPS description of the operator. There is a $ prefix on the DELETE function because this function is user-defined: Its definition is the value of the variable DELETE. The statement (ASSERT (STATUS $M ON)) represents the add list of the operator. The final statement, ($BUILD (′ (:$TURNONLIGHT ACTION $M))), simply adds the action of turning on the light to the planned sequence of actions to be carried out by the robot.

DESIGN PHILOSOPHY REVISITED

Although our operators are as concise as those of STRIPS, we have given them a certain amount of strategic information. For example, in *turnonlight* we tell the system that the box must be brought up to the light switch *before* the robot mounts the box, while in STRIPS the same preconditions are unordered, so the planner may investigate the ill-advised possibility of climbing the box first and moving it later. STRIPS could have been given ordered preconditions, but its designers were more interested in the behavior of the problem solver when it had to discover the best ordering by itself. The decision about how many hints to give the operator is, we feel, primarily a matter of taste. For QA4 we prefer to give the operators as much information as possible, and risk the charge of using an ad hoc approach. We feel that this is the only way that our programs will solve interesting problems.

Although STRIPS does not rely as heavily on axiom-based theorem proving as QA3 does, it still uses a theorem prover for such purposes as determining whether an operator is relevant or applicable, tasks that QA4 accomplishes by using pattern matching. As STRIPS has shown us, the theorem proving involved in such processes is quite straightforward, and a pattern-matcher seems to be a more appropriate tool here than a full-fledged theorem prover.

We also applied the system to the two other problems from the STRIPS paper.[9] In these problems the robot is envisioned to be in a building with several rooms and a corridor. It is asked first to push together the three boxes in one of the rooms and second to find its way from one of the rooms to another. The QA4 system solved these problems also, and it made no mistaken choices.

These problems are, of course, particularly simple. Had they been sufficiently complicated, any QA4 program would have to do some searching in trying to find a solution. In that case, we would have had to write our operators in a somewhat different way.

OTHER FEATURES AND APPLICATIONS

These problems did not use many of the features of QA4 that we feel would be valuable for more complex problems and in other problem domains. For example, STRIPS plans are always linear sequences of operators; plans never include branches that prepare for various contingencies. There is no mechanism for considering alternative world models in a single plan. The construction of conditional plans is facilitated by the "context mechanism" of QA4, which allows us to store alternative hypotheses under distinct contexts without confusion.

STRIPS plans also have no loops; an action in a STRIPS plan can be repeated only a prespecified number of times. However, the fact that QA4 plans are programs that admit both interaction and recursion opens the possibility of writing plans with repeated actions.

If QA4 is successful in writing robot plans with loops, it will probably be equally effective at the synthesis of computer programs. Assembly code programs, in particular, are strikingly similar to robot plans: Computer instructions are analogous to operators, and whereas for robot plans we model the world, for computer programs we model the state of the registers of the machine. QA4 has already been successful at producing simple straight-line assembly code programs. More general theorem-proving ability would enable QA4 to construct programs in other languages, including QA4 itself.

We also plan to apply QA4 to the verification of existing programs.[10] For this application we need considerable sophistication in formula manipulation and the handling of arithmetic relations. We have introduced some new data types, sets, and bags (bags are like sets, but may have several instances of the same element), which simplify many arithmetic problems. For example, a stumbling block of earlier deductive systems has been the equality relation, which has required either a plethora of new axioms or a slightly less clumsy new rule of inference in order to describe its properties. In QA4 we simply place expressions known to be equal in the same set; the symmetric, reflexive, and transitive laws then follow from the properties of sets, and need not be stated explicitly.

A similar technique simplifies the description of commutative functions of n arguments, such as plus and times. We make these arguments bags rather than n-tuples. Then the commutative law for addition need no longer be mentioned, since bags, like sets, are unordered.

PLANNER AND QA4

Let us examine the similarities and differences between QA4 and PLANNER. The two languages are quite similar in conception, and there are also more detailed parallels. Our operators in the above example are similar to PLANNER consequent theorems. ASSERT, GOAL, and EXISTS all have their PLANNER counterparts. Both the variable prefixes and the heavy reliance on pattern matching are PLANNER features. PLANNER also has built-in backtracking.

There are some differences in detail. QA4 relies more on the use of new data types such as sets, whereas PLANNER would implement the same features by using more complex procedures. The context mechanism is unique to QA4, and a coroutine mechanism has been implemented in QA4 but is only projected in PLANNER. The PLANNER pattern matcher that has been implemented does not allow the nesting of patterns. Moreover, QA4 is intended to be modified in design and implementation, with experience. Therefore it is implemented in LISP, and we have no immediate plans to rewrite it in assembly code; PLANNER is being implemented in assembly code.

IMPLEMENTATION

The BBN-LISP system[11] in which QA4 is embedded has many sophisticated debugging features, and QA4 has been designed to take advantage of such packages as the BBN editor and the programmer's assistant. At the top level, QA4 and LISP coexist side by side: Lines preceded by ! go to the QA4 evaluator rather than the LISP interpreter.

Although we have never stressed efficiency in our design, we have managed to make the implementation reasonably efficient. Our representation of the robot problems compares favorably with that of STRIPS: The solutions to the three problems were each found in less than half a minute. The TRACE feature, whose output is shown in the appendix, is quite elaborate; we can follow the search for a solution with the appropriate degree of detail.

The ease with which it was possible to construct a robot planner on this scale gives us hope that QA4 will be an appropriate vehicle for the development of more complex problem solvers.

ACKNOWLEDGMENTS

The research reported herein was sponsored by the National Aeronautics and Space Administration under Contract NASW-2086.

The goals and original concepts of QA4 were formulated with C. Cordell Green and Robert A. Yates. We have benefited by discussions with and criticism from other members of the staff of the Artificial Intelligence Center at SRI, especially Bertram Raphael and the STRIPS group, Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. We have also profited from many conversations with Carl Hewitt. Nils J. Nilsson and Bertram Raphael gave us critical readings of the manuscript.

REFERENCES

1 J A C DERKSEN
The QA4 primer
Artificial Intelligence Center Stanford Research Institute
Menlo Park California 1972

2 J F RULIFSON
QA4 programming concepts
Artificial Intelligence Technical Note 60
Artificial Intelligence Center Stanford Research Institute
Menlo Park California 1971

3 T A WINOGRAD
Procedures as a representation for data in a computer program for understanding natural language
Ph.D. Thesis Department of Mathematics Massachusetts Institute of Technology Cambridge Massachusetts

4 R W FLOYD
Assigning meanings to programs
Mathematical Aspects of Computer Science Volume 19
American Mathematical Society pp 19-32 Providence Rhode Island 1967

5 Z MANNA R WALDINGER
Towards automatic program synthesis
CACM Volume 14 pp 151-165 1971

6 W W BLEDSOE R S BOYER W H HENNEMAN
Computer proofs of limit theorems
Artificial Intelligence Volume 3 pp 27-68 1972

7 R E FIKES N J NILSSON
STRIPS: a new approach to the application of theorem proving to problem solving
Artificial Intelligence Volume 2 pp 289-308 1971

8 C C GREEN
Application of theorem proving to problem solving
International Joint Conference on Artificial Intelligence Washington 1969

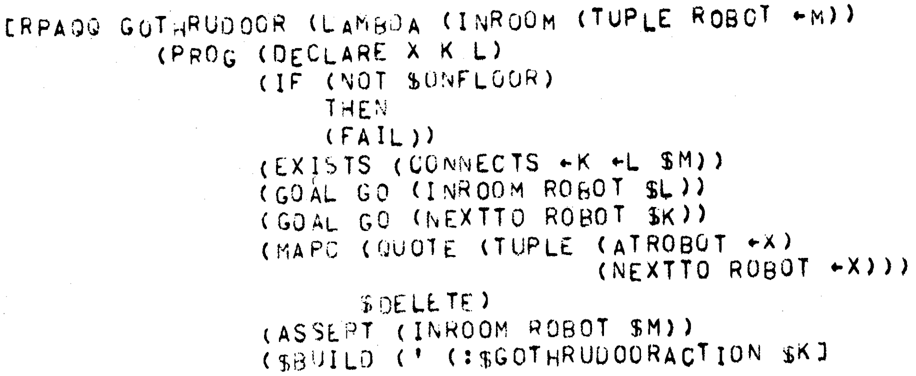
9 C HEWITT
Description and theoretical analysis (using schematic) of PLANNER: a language for proving theorems and manipulating model in a robot
Ph.D. thesis Department of Mathematics Massachusetts Institute of Technology Cambridge Massachusetts 1972

10 B ELSPAS M W GREEN K N LEVITT R J WALDINGER
Research in interactive program-proving techniques
Information Science Laboratory Stanford Research Institute Menlo Park California

11 W TEITELMAN D G BOBROW A K HARTLEY D L MURPHY
BBN-LISP TENEX reference manual
Cambridge Massachusetts 1972

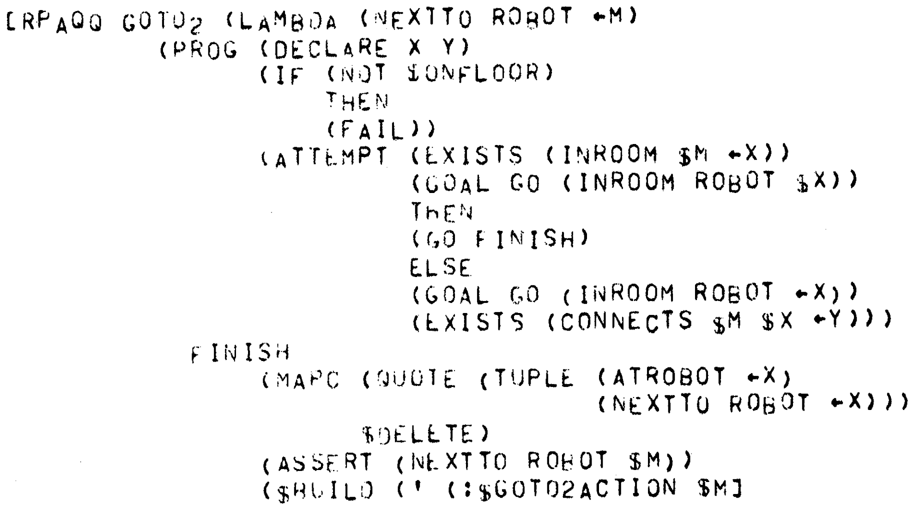
APPENDIX

THE QA4 OPERATORS: THE GOTHRUDOOR OPERATOR



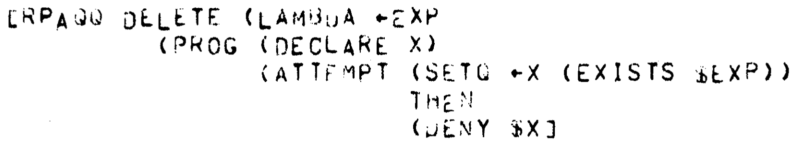
```
[RPAQQ GOTHRUDOOR (LAMBDA (INROOM (TUPLE ROBOT ←M))
        (PROG (DECLARE X K L)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (EXISTS (CONNECTS ←K ←L $M))
              (GOAL GO (INROOM ROBOT $L))
              (GOAL GO (NEXTTO ROBOT $K))
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (INROOM ROBOT $M))
              ($BUILD (' (:$GOTHRUDOORACTION $K]
```

THE GOTO2 OPERATOR



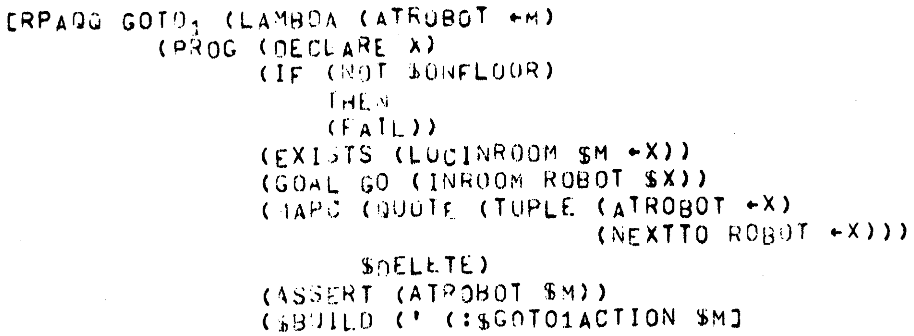
```
[RPAQQ GOTO2 (LAMBDA (NEXTTO ROBOT ←M)
        (PROG (DECLARE X Y)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (ATTEMPT (EXISTS (INROOM $M ←X))
                       (GOAL GO (INROOM ROBOT $X))
                       THEN
                       (GO FINISH)
                       ELSE
                       (GOAL GO (INROOM ROBOT ←X))
                       (EXISTS (CONNECTS $M $X ←Y)))
         FINISH
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (NEXTTO ROBOT $M))
              ($BUILD (' (:$GOTO2ACTION $M]
```

THE FUNCTION DELETE



```
[RPAQQ DELETE (LAMBDA ←EXP
        (PROG (DECLARE X)
              (ATTEMPT (SETQ ←X (EXISTS $EXP))
                       THEN
                       (DENY $X]
```

THE GOTO1 OPERATOR



```
[RPAQQ GOTO1 (LAMBDA (ATROBOT ←M)
        (PROG (DECLARE X)
              (IF (NOT $ONFLOOR)
                  THEN
                  (FAIL))
              (EXISTS (LOCINROOM $M ←X))
              (GOAL GO (INROOM ROBOT $X))
              (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                  (NEXTTO ROBOT ←X)))
                    $DELETE)
              (ASSERT (ATROBOT $M))
              ($BUILD (' (:$GOTO1ACTION $M]
```

THE PUSHTO OPERATOR

```
[RPAQQ PUSHTO (LAMBDA (NEXTTO ←M ←N)
         (PROG (DECLARE X)
               (IF (NOT $ONFLOOR)
                   THEN
                   (FAIL))
               (EXISTS (PUSHABLE $M))
               (ATTEMPT (EXISTS (INROOM $M ←X))
                        (EXISTS (INROOM $N $X))
                        THEN
                        (GO FINISH)
                        ELSE
                        (EXISTS (INROOM $M ←X))
                        (EXISTS (CONNECTS $N $X ←Y)))
           FINISH
               (GOAL GO (NEXTTO ROBOT $M))
               (MAPC (QUOTE (TUPLE (ATROBOT ←X)
                                   (AT $M ←X)
                                   (NEXTTO ROBOT ←X)
                                   (NEXTTO $M ←X)
                                   (NEXTTO $X $M)))
                     $DELETE)
               (ASSERT (NEXTTO $M $N))
               (ASSERT (NEXTTO $N $M))
               (ASSERT (NEXTTO ROBOT $M))
               ($BUILD (' (:$PUSHTOACTION (TUPLE $M $N]
```

THE CLIMBONBOX OPERATOR

```
[RPAQQ CLIMBONBOX (LAMBDA (ON ROBOT ←M)
         (PROG (DECLARE X)
               (IF (NOT $ONFLOOR)
                   THEN
                   ($CLIMBOFFBOX))
               (EXISTS (TYPE $M BOX))
               (GOAL GO (NEXTTO ROBOT $M))
               ($DELETE (QUOTE (ATROBOT ←X)))
               (SETQ ←ONFLOOR FALSE)
               (ASSERT (ON ROBOT $M))
               ($BUILD (' (:$CLIMBONBOXACTION $M]
```

THE CLIMBOFFBOX OPERATOR

```
[RPAQQ CLIMBOFFBOX (LAMBDA (TUPLE)
         (PROG (DECLARE M)
               (EXISTS (ON ROBOT ←M))
               (EXISTS (TYPE $M BOX))
               ($DELETE (QUOTE (ON ROBOT $M)))
               (SETQ ←ONFLOOR TRUE)
               ($BUILD (' (:$CLIMBOFFBOXACTION $M]
```

THE TURNONLIGHT OPERATOR

```
[RPAQQ TURNONLIGHT (LAMBDA (STATUS ←M ON)
         (PROG (DECLARE N)
               (EXISTS (TYPE $M LIGHTSWITCH))
               (EXISTS (TYPE ←N BOX))
               (GOAL DO (NEXTTO $N $M))
               (GOAL DO (ON ROBOT BOX1))
               ($DELETE (QUOTE (STATUS $M OFF)))
               (ASSERT (STATUS $M ON))
               ($BUILD (' (:$TURNONLIGHTACTION $M]
```

THE FUNCTION BUILD

```
  [RPAQQ BUILD (LAMBDA ←X
           (SETQ ←ANSWER (CONS $X $ANSWER]
```

THE FUNCTION SOLVE

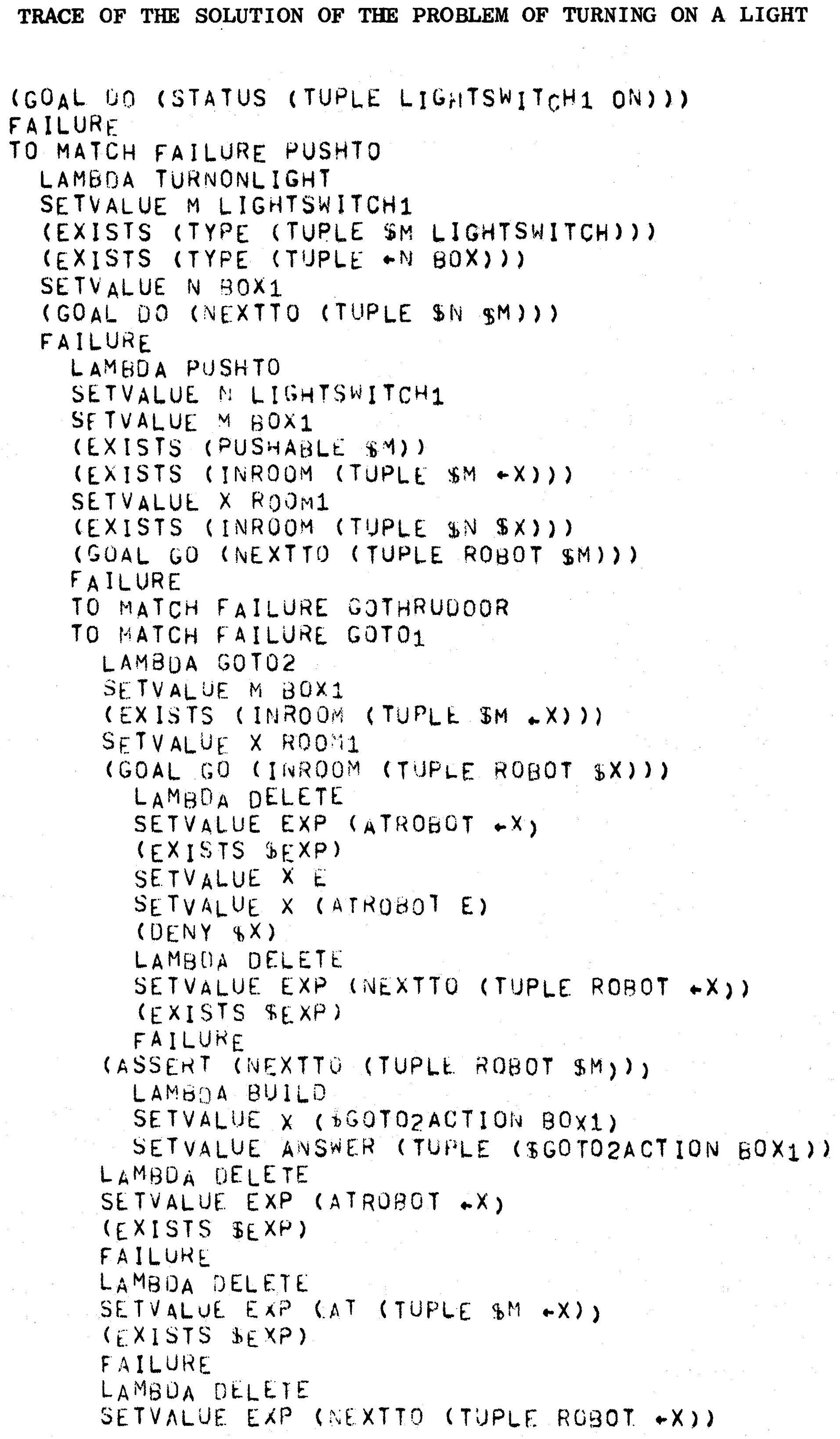
```
  [RPAQQ SOLVE (LAMBDA ←PROBLEM
           (PROG (DECLARE X)
                 (SETQ ←X (REVERSE $PROBLEM))
                 (RETURN (PROG (DECLARE)
                               $$X ]
```

The model of the robot world. Expressions are evaluated by the QA4 evaluator ("!") and stored in the net.

```
(DEFLIST(QUOTE(
  [SETUP (((! (TO DO CLIMBONBOX)))
          ((! (TO DO TURNONLIGHT)))
          ((! (TO DO PUSHTO)))
          ((! (TO GO GOTHRUDOOR)))
          ((! (TO GO GOTO1)))
          ((! (TO GO GOTO2)))
          ((! (SETQ ←ONFLOOR TRUE)))
          [(! (SETQ ←ANSWER (' (TUPLE]
          [(! (ASSERT (INROOM LIGHTSWITCH1 ROOM1]
          [(! (ASSERT (INROOM ROBOT ROOM1]
          [(! (ASSERT (ATROBOT E]
          [(! (ASSERT (LOCINROOM F ROOM4]
          [(! (ASSERT (PUSHABLE BOX1]
          [(! (ASSERT (PUSHABLE BOX2]
          [(! (ASSERT (PUSHABLE BOX3]
          [(! (ASSERT (INROOM BOX1 ROOM1]
          [(! (ASSERT (INROOM BOX2 ROOM1]
          [(! (ASSERT (INROOM BOX3 ROOM1]
          [(! (ASSERT (STATUS LIGHTSWITCH1 OFF]
          [(! (ASSERT (TYPE LIGHTSWITCH1 LIGHTSWITCH]
          [(! (ASSERT (TYPE BOX1 BOX]
          [(! (ASSERT (TYPE BOX2 BOX]
          [(! (ASSERT (TYPE BOX3 BOX]
          [(! (ASSERT (AT LIGHTSWITCH1 D]
          [(! (ASSERT (AT BOX1 A]
          [(! (ASSERT (AT BOX2 B]

          [(! (ASSERT (AT BOX3 C]
          [(! (ASSERT (CONNECTS DOOR1 ROOM1 ROOM5]
          [(! (ASSERT (CONNECTS DOOR1 ROOM5 ROOM1]
          [(! (ASSERT (CONNECTS DOOR2 ROOM2 ROOM5]
          [(! (ASSERT (CONNECTS DOOR2 ROOM5 ROOM2]
          [(! (ASSERT (CONNECTS DOOR3 ROOM3 ROOM5]
          [(! (ASSERT (CONNECTS DOOR3 ROOM5 ROOM3]
          [(! (ASSERT (CONNECTS DOOR4 ROOM4 ROOM5]
          ((! (ASSERT (CONNECTS DOOR4 ROOM5 ROOM4]
```

THE 3 INITIAL PROBLEM STATEMENTS

```
  [3BOXPROBLEM (((! ($SOLVE (LIST (GOAL DO (NEXTTO BOX1 BOX2))
                                  (GOAL DO (NEXTTO BOX2 BOX3]
  [GOROOMPROBLEM (((! ($SOLVE (GOAL GO (ATROBOT F]
  [TURNONLIGHTPROBLEM (((! ($SOLVE (GOAL DO (STATUS LIGHTSWITCH1 ON]
))(QUOTE HISTORY))
```



TRACE OF THE SOLUTION OF THE PROBLEM OF TURNING ON A LIGHT

```
(GOAL DO (STATUS (TUPLE LIGHTSWITCH1 ON)))
FAILURE
TO MATCH FAILURE PUSHTO
  LAMBDA TURNONLIGHT
  SETVALUE M LIGHTSWITCH1
  (EXISTS (TYPE (TUPLE $M LIGHTSWITCH)))
  (EXISTS (TYPE (TUPLE ←N BOX)))
  SETVALUE N BOX1
  (GOAL DO (NEXTTO (TUPLE $N $M)))
  FAILURE
    LAMBDA PUSHTO
    SETVALUE N LIGHTSWITCH1
    SETVALUE M BOX1
    (EXISTS (PUSHABLE $M))
    (EXISTS (INROOM (TUPLE $M ←X)))
    SETVALUE X ROOM1
    (EXISTS (INROOM (TUPLE $N $X)))
    (GOAL GO (NEXTTO (TUPLE ROBOT $M)))
    FAILURE
    TO MATCH FAILURE GOTHRUDOOR
    TO MATCH FAILURE GOTO1
      LAMBDA GOTO2
      SETVALUE M BOX1
      (EXISTS (INROOM (TUPLE $M ←X)))
      SETVALUE X ROOM1
      (GOAL GO (INROOM (TUPLE ROBOT $X)))
        LAMBDA DELETE
        SETVALUE EXP (ATROBOT ←X)
        (EXISTS $EXP)
        SETVALUE X E
        SETVALUE X (ATROBOT E)
        (DENY $X)
        LAMBDA DELETE
        SETVALUE EXP (NEXTTO (TUPLE ROBOT ←X))
        (EXISTS $EXP)
        FAILURE
      (ASSERT (NEXTTO (TUPLE ROBOT $M)))
        LAMBDA BUILD
        SETVALUE X ($GOTO2ACTION BOX1)
        SETVALUE ANSWER (TUPLE ($GOTO2ACTION BOX1))
      LAMBDA DELETE
      SETVALUE EXP (ATROBOT ←X)
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (AT (TUPLE $M ←X))
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE ROBOT ←X))
      (EXISTS $EXP)
      SETVALUE X BOX1
      SETVALUE X (NEXTTO (TUPLE ROBOT.BOX1))
      (DENY $X)
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE $M ←X))
      (EXISTS $EXP)
      FAILURE
      LAMBDA DELETE
      SETVALUE EXP (NEXTTO (TUPLE $X $M))
      (EXISTS $EXP)
      FAILURE
    (ASSERT (NEXTTO (TUPLE $M $N)))
    (ASSERT (NEXTTO (TUPLE $N $M)))
    (ASSERT (NEXTTO (TUPLE ROBOT $M)))
      LAMBDA BUILD
      SETVALUE X ($PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
      SETVALUE ANSWER (TUPLE ($PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
($GOTO2ACTION BOX1))
  (GOAL DO (ON (TUPLE ROBOT BOX1)))
  FAILURE
  TO MATCH FAILURE PUSHTO
  TO MATCH FAILURE TURNONLIGHT
    LAMBDA CLIMBONBOX
    SETVALUE M BOX1
    (EXISTS (TYPE (TUPLE $M BOX)))
    (GOAL GO (NEXTTO (TUPLE ROBOT $M)))
      LAMBDA DELETE
      SETVALUE EXP (ATROBOT ←X)
      (EXISTS $EXP)
      FAILURE
    SETVALUE ONFLOOR FALSE
    (ASSERT (ON (TUPLE ROBOT $M)))
      LAMBDA BUILD
      SETVALUE X ($CLIMBONBOXACTION BOX1)
      SETVALUE ANSWER (TUPLE ($CLIMBONBOXACTION BOX1) ($PUSHTOACTION
(TUPLE BOX1 LIGHTSWITCH1)) ($GOTO2ACTION BOX1))
    LAMBDA DELETE
    SETVALUE EXP (STATUS (TUPLE $M OFF))
    (EXISTS $EXP)
    FAILURE
  (ASSERT (STATUS (TUPLE $M ON)))
    LAMBDA BUILD
    SETVALUE X ($TURNONLIGHTACTION LIGHTSWITCH1)
    SETVALUE ANSWER (TUPLE ($TURNONLIGHTACTION LIGHTSWITCH1) (
$CLIMBONBOXACTION BOX1) ($PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
($GOTO2ACTION BOX1))
  LAMBDA SOLVE
  SETVALUE PROBLEM (TUPLE ($TURNONLIGHTACTION LIGHTSWITCH1) (
$CLIMBONBOXACTION BOX1) ($PUSHTOACTION (TUPLE BOX1 LIGHTSWITCH1))
($GOTO2ACTION BOX1))
  SETVALUE X (TUPLE ($GOTO2ACTION BOX1) ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($CLIMBONBOXACTION BOX1) ($TURNONLIGHTACTION LIGHTSWITCH1))
```

THE ANSWER RETURNED BY QA4

```
(PROG (DECLARE) ($GOTO2ACTION BOX1) ($PUSHTOACTION (TUPLE BOX1
LIGHTSWITCH1)) ($CLIMBONBOXACTION BOX1) ($TURNONLIGHTACTION LIGHTSWITCH1))
```

Recent developments in SAIL—An ALGOL-based language for artificial intelligence

by J. A. FELDMAN, J. R. LOW, D. C. SWINEHART and R. H. TAYLOR

Stanford University
Stanford, California

INTRODUCTION

Progress in Artificial Intelligence has traditionally been accompanied by advances in special purpose programming techniques and languages. Virtually all of this development has been concentrated in languages and systems oriented to list processing. As the efforts of Artificial Intelligence researchers began to turn from purely symbolic problems toward interaction with the real world, certain features of algebraic languages became desirable. There were several attempts (notably LISP2 and FORMULA ALGOL) to combine the best features of both kinds of language. At the same time, designers of algebraic languages began to include features for non-numerical computation. No new general purpose language without some sort of list processing facility has been suggested for several years. We have followed a tack somewhat different from either of these in the design of SAIL and in its subsequent modifications.

The starting point for the development of SAIL was the recognized need for a language incorporating symbolic and algebraic capabilities, primarily for Hand-Eye research. The problems are somewhat similar to those in Computer Graphics and one of us had just developed a language, LEAP,[4] for such applications. After an attempt to honestly evaluate alternative techniques, we decided that the associative processing features of LEAP were the way to go. There are important differences between LEAP and the first SAIL (primarily in input-output, string manipulation, and implementation), but these differences are not relevant here. It is essentially this system for the PDP-10 which is distributed by DECUS and is being used for Artificial Intelligence and other research in a number of laboratories.

This original SAIL met our needs for about two years before requiring serious change. Then we began to face the problem of putting together a hand-eye system which was much bigger than the available main memory and which did not lend itself to a static overlay structure. Our solution involves a number of language additions which facilitate the treatment of jobs under the time-sharing system as a set of cooperating sequential processes, and has been described in Reference 5. The three main additions were: a monitor for user control and debugging, a shared data facility, and the introduction of message procedures. The shared data facility makes use of the second relocation register of the PDP-10 to allow jobs to access a common global data area in a natural and efficient manner. The message procedures are the main mechanism for asynchronous communication and control between jobs. A message procedure is a procedure in one job which can be invoked from another job. Control information associated with the invocation can provide the effect of subroutines, coroutines, parallel processes, events, and a variety of other disciplines. These multitasking modifications to SAIL have enabled researchers to assemble and modify large collections of jobs with a minimum amount of attention to system problems.

A number of factors have combined recently to cause us to make a second set of major modifications to SAIL. The multi-tasking facilities of the second SAIL were seen to be at least as useful within a single job as they were across jobs. In addition, the ability to assemble large collections of routines brought us to the point of facing one of the core problems of Artificial Intelligence—what is the right sequence of actions for carrying out a given task in a particular environment. This strategy problem is currently very popular and is the driving force behind much of the recent development in languages for Artificial Intelligence. Our view of the problem is somewhat unorthodox and merits some discussion.

Problem solving for an entity which deals with the real world is fraught with uncertainty. The state of the world cannot be assumed to be known—in fact, one of the main goals of a strategy must be to gain enough information to carry out the task. An additional problem arises in resource allocation; even if an exhaustive search of the environment will yield a solution, it may not do so at an acceptable cost. Considerations of this sort cause us to view the strategy problem as inherently involving numerical estimates of probabilities, costs, etc. A complete discussion of these issues is beyond the scope of this paper, but the recent SAIL modifications have been influenced by our model of the strategy problem.

Our recent language work has been intended to facilitate the design of programs for the construction and execution of strategies for interaction with the real world. The facilities are being applied to other problems, but we will concentrate on the original theme. However, the language design effort was concerned with expanding the power of SAIL as a general purpose language as opposed to developing a special purpose system. One critical design constraint was that the features not entail large hidden overheads or appreciably degrade the performance of programs not making use of them. We believe we have found a set of features which meet our design goals. The major additions are: backtracking, procedure variables, matching procedures, and a general multi-tasking facility.

STATE SAVING AND BACKUP

In order to try several different alternative strategies it is often necessary to save the current state of the computation. Thus, if the first attempt does not succeed, we may "back up" and try one of the other alternatives. We may also switch between alternatives, continuing with one only until it no longer seems the most promising, but retaining the option of resuming it later if the other alternatives do not prove to be satisfactory. Another technique used in programming non-deterministic algorithms, parallel processes, will be discussed later in this paper.

In general the state of a SAIL computation includes the current control environment, the input and output which have been requested, the contents of the LEAP associative store and the contents of all variables. New SAIL has features which will help handle the last of these components: the contents of variables.

We normally do not want to have the values of all variables "backed-up" when we switch between alternatives. One reason is that it is often useful for one alternative to communicate certain pieces of information it has acquired to the other alternatives. This information is usually saved in certain variables. If we back-up those variables, we lose the information. Another reason for not backing-up all variables is that often only a small subset will have meaning for more than a single alternative, and it is very costly to back-up large amounts of data which may not be relevant for the other alternatives. Therefore we have implemented ways of saving the values of specific variables and then restoring them at a later time.

The state-saving mechanism is based on two new statements: REMEMBER, and RESTORE. Each of these operate on a new SAIL data-type called a "context." A context consists of a set of references to variables and their values.

We save the contents of variables by means of REMEMBER statements,

```
REMEMBER (i, j, a[3]) IN context1:
```

This statement would save the values of "i," "j," "a[3]" in the context named "context1." If any of these variables had been previously saved in "context1," the old values would be lost.

An alternate form of the REMEMBER statement is:

```
REMEMBER ALL IN context1;
```

The current value of each variable which has been remembered in "context1" would replace the value that was previously stored there.

The RESTORE statement also has two forms. The first has an argument-list,

```
RESTORE (j, a[3]) FROM context1;
```

This would search context1 for the arguments and give an error indication if any were not "remembered" within that context. The values saved for those arguments "remembered," would be restored to the appropriate variables.

The other form of the RESTORE statement is:

```
RESTORE ALL FROM context1;
```

This would restore the contents of all variables saved within the named context.

These new features seem to provide the most important features of state-saving without the large overhead imposed by automatic back-up of the entire state or incremental state-saving as implemented in some other programming systems.

LEAP

SAIL contains an associative data system called LEAP which is used for symbolic computations. LEAP is a combination of syntax and runtime subroutines for handling items, sets of items and associations.

An item is similar to a LISP atom. Items may be declared or obtained during execution from a pool of items by using the function NEW. Items may be stored in variables (itemvars), be members of sets, be elements of lists, or be associated together to form triples (associations) within the associative store.

A set is an unordered collection of distinct items. Items may be inserted into set variables by "PUT" statements and removed from set variables by "REMOVE" statements. Set expressions may also be assigned to set variables. The simplest set expression is of the form:

{item1, item2, item3 . . .}

which represents the set consisting of the denoted items. More complicated set expressions involving set functions, set union, subtraction and intersection are also provided. Sets are stored in a canonical internal form which allows us to carry out such operations as intersection, union and comparison in a time proportional to the lengths of the sets involved.

Sets are deficient in some applications, though, because they are unordered. Thus we could not easily try different alternatives in order of their expected utility. To remedy this, as well as provide a mechanism for creation of parameter lists to interpretively called procedures (see PROCEDURE VARIABLES below), SAIL now contains a data-type called "list." A list is simply an ordered sequence of items. An item may appear more than once within a list. List operations include inserting and removing specific items from a list variable by indexed PUT and REMOVE statements. List variables may also be assigned list expressions, the simplest of which is of the form;

{{item1, item2, item3 . . .}}

which represents the explicit sequence of denoted items. Other list expressions include list functions, concatenation, and sublists.

Triples are ordered three-tuples of items, and may themselves be considered items and occur in subsequent associations. They are added to the associative store by executing MAKE statements. For example:

MAKE use ⊗ plan1 ≡ task1;

The three item components of an association are refered to as the "attribute," the "object," and the "value," respectively. Associations may be removed from the store by using ERASE statements such as:

ERASE use ⊗ plan1 ≡ ANY;

Each item other than those representing associations may have a DATUM which is a scalar or array of any SAIL data-type. The data-type of a DATUM may be checked during execution. DATUMs are used much as variables. For example:

DATUM(it) ← 5;

would cause the datum of the item "it" to be replaced with "5."

SAIL contains a compile-time macro facility which allows such things as string substitution and conditional compilation. As is the custom of many SAIL programmers, we will use the macro "∂" to stand for the string "DATUM." Thus the above example would appear as:

∂(it) ← 5;

PROCEDURE VARIABLES

It is quite natural in an interpreter to allow for the execution of program generated sequences of actions. This is an important feature for artificial intelligence applications and is not easily made available for compiled programs. In new SAIL, the generation of such sequences is facilitated by a procedure variable mechanism which fits in quite nicely with the associative search features of the language. These procedure variables are created at runtime from items by statements of the form

ASSIGN(⟨item expression⟩, ⟨procedure specification⟩)

where

⟨procedure specification⟩ : : = ⟨procedure id⟩ |
DATUM (⟨procedure item expression⟩)

For instance,

ASSIGN(xxx, baz)

would cause the datum of item xxx to contain a description of baz, together with a pointer to baz's current environment. Similarly, the statement

ASSIGN(yyy, ∂(xxx))

would cause yyy to be made into a procedure item containing the same information as that in xxx.

In addition to dynamically specifying what procedure to execute, one would also like a convenient way to dynamically specify an argument list for a procedure call. This facility is provided by the APPLY mechanism:

APPLY(⟨procedure specification⟩, ⟨argument list⟩)

where ⟨argument list⟩ is any SAIL list and may be omitted if the procedure has no parameters. For example,

```
APPLY(foo)
APPLY(∂(xxx), list1)
APPLY(∂(APPLY(yyy)), {{x, y, z}})
```

APPLY uses the items in the argument list, together with the environment information from the procedure item (or from the current environment, if the procedure is named explicitly) to make the appropriate procedure call. If the called procedure produces a value, that value will be returned as the value of APPLY.

Procedure items permit a great deal of flexibility. For instance, the user can say things like

```
FOREACH x | xεactions ∧ use⊗x≡fastening do
  BEGIN
  APPLY(∂(x), {{board1, board2}});
  IF together(board1, board2) THEN
  GO TO done_it;
  END;
done_it:
```

This would search the set "actions" for any procedures which have been asserted to be useful for fastening things together until either the list is exhausted or the task is successfully completed.

MULTIPLE PROCESSES

The control structure of SAIL was originally very much like that of Algol 60—that is to say block structured and procedure oriented. Although this structure is adequate for many problems, there are some cases in which it is uncomfortably restrictive. In hand-eye applications, for instance, there are frequently modules of code which are more or less mutually independent but that wish to call on each other for various services. Similarly, one may wish to investigate several possible strategies at once, with the results of one computation perhaps influencing the course of others. In such cases, it is much more natural to think of (and write) these modules as coroutines or independent processes rather than as nested procedure calls. To some extent, message procedures provided the desired facilities, with each job acting as a separate process. This solution has some rather severe drawbacks, since the overhead involved in switching control from process to process and in interprocess communication is so high that close interaction becomes prohibitively expensive. One of our goals in providing new control facilities was to make possible the close cooperation of many small-to medium-sized processes within a single job without imposing an excessive overhead either on old-style procedural programs or on users of the shiny new features. In doing this, we wanted to retain the block structure rules of Algol, since these rules are generally familiar to programmers and provide a useful means of determining which data is to be shared.

The implementation we have chosen somewhat resembles the mechanism described by Organick & Cleary[8] for the Burroughs B6700. In SAIL, a process is essentially a procedure activation which has been given its own run time stack and which thus does not have to return before the process that invoked it can continue. SAIL procedures normally make up-level references via a "static" (lexical nesting) chain maintained for that purpose in the stack. When a procedure is to be called as an independent process, a "process" routine first gets space for a new stack. It then sets up appropriate process control variables in the new stack area and in the "parent." Finally, the procedure is invoked using the new stack. When this procedure is entered, it will set up its static link by looking back along the static chain of the calling process until it finds an activation of its lexical parent. Thus, different processes will share data belonging to their common ancestors.

Many of the applications which we have considered do not permit us to predict just how many subprocesses a process might wish to spawn or require that several processes be instantiated using the same procedure on different data. Therefore, we have chosen to "name" processes by assigning them to LEAP items, rather than by using procedure names or some special data type called "process." This approach has the added advantage of allowing complex structures of processes to be built up using the mechanisms of LEAP. New processes are created by statements of the form:

```
SPROUT(⟨item expression⟩, ⟨procedure call⟩,
    ⟨options⟩)
```

where the item specified by ⟨item expression⟩ is to be used as the process name, the ⟨procedure call⟩ tells what this process is to do, and ⟨options⟩ is an integer which is used to specify how certain process attributes are to be set up. (If the ⟨options⟩ parameter is omitted or only partially specified, SAIL will provide default values). For instance, a procedure to nail two boards together might contain a sequence like

```
:
ITEM p1, p2, p3;
:
:
SPROUT (p1, grab (hand1, hammer));
SPROUT (p2, grab (hand2, nail));
SPROUT (p3, lookat (tv1, boards));
:
:
JOIN ({p1, p2, p3});
pound (hammer, nail, boards);
:
:
```

In this case, grab(hand1, hammer) would be executed as process p1, grab(hand2, nail) would be executed as process p2, and lookat(tv1, boards) would be executed as process p3. The process creating them continues on its way down to the JOIN statement. In general,

JOIN(⟨set⟩)

causes the process executing it to be suspended until all the processes named by the ⟨set⟩ have terminated. Thus pound (hammer, nail, boards) will not be called until p1, p2, and p3 have all terminated. In our example, both SPROUTed processes and the original process would theoretically run in parallel. In fact, this is not possible with a single processor. Instead, the SAIL runtime system includes a scheduler that decides which process is to be executed at any given instant. Each process is given a priority and time quantum and may be in one of four states: "running," "ready" (i.e., runnable), "suspended," or "terminated." The scheduler, which is invoked either by a clock interrupt or by an explicit call by the user, uses a simple round robin algorithm to distribute service among the highest priority ready processes.

When a process is SPROUTed, the system assigns it a standard default priority and time quantum, unless the user specifies otherwise by appropriate options. The SPROUTed process usually becomes the running process, while the SPROUTing process reverts to ready status, unless some other option is specified. For instance, suppose we have some procedure "wander" which searches a data base or the real world at random for potentially useful objects. Then we might write something like:

```
SPROUT(wanderer←NEW, wander(world_model),
  PRIORITY(very_low)+QUANTUM(2)
  +RUN_ME)
```

The current process would continue to run, and wanderer would languish in ready status until everything of higher priority had been suspended.

Processes may be suspended or terminated via

SUSPEND(⟨process item expression⟩)

and

TERMINATE(⟨process item expression⟩)

which do just what one might expect. Similarly, SAIL provides system functions for changing a process' priority or quantum.

Co-routine style interactions are facilitated by the use of the RESUME construct:

```
x←RESUME(⟨process item expression⟩,
  ⟨return value⟩, ⟨options⟩)
```

where ⟨options⟩ is again optional. The usual effect of RESUME is to cause the currently running process to be suspended and the process specified by ⟨process item expression⟩ to become running. If the process being resumed had suspended itself by means of a resume statement, then it will receive ⟨return value⟩ as the value of the RESUME. For instance,

```
:
PROCEDURE tool_getter(ITEMVAR tool_type);
  BEGIN
  ITEMVAR tool;
  FOREACH tool | tool ∈ tool_box ∧ type⊗tool
    ≡tool_type DO RESUME (CALLER(THIS_
    PROCESS), tool);
  END;
:
SPROUT(tg←NEW, tool_getter (screwdriver),
  SUSPEND_HIM),
DO sd←RESUME(tg, NIC) UNTIL
  fits(sd, screw1);
TERMINATE(tg);
:
```

In this case, the tool getter process "tg" will be initialized and immediately suspended. Then, the RESUME (tg, NIC) will wake it up to find one screwdriver, which will be assigned to itemvar "sd" by the RESUME (CALLER (THIS_PROCESS), tool). (THIS_PROCESS and CALLER (⟨procid⟩) are system supplied routines that return the process items for the currently running process and for the process that last awakened process ⟨procid⟩, respectively). Later on, we will discuss a somewhat cleaner solution, using matching procedures, to the problem used for this illustration. We will also show how the interprocess communication facilities of the language may be used to handle the problem of what to do if tool_getter runs out of tools.

FOREACH STATEMENTS

The standard way of searching the LEAP associative store is the FOREACH statement. A FOREACH statement consists of a "binding list" of itemvars, an "associative context" and a statement to be iterated. Consider the following example,

```
FOREACH gp, p, c | parent ⊗ c ≡ p ∧ parent
   ⊗ p ≡ gp DO MAKE grandparent ⊗ c ≡ gp;
```

In this example the binding-list consists of the itemvars "gp," "p," "c." The associative context consists of two "elements," "parent ⊗ c ≡ p," and "parent ⊗ p ≡ gp." The statement to be iterated is the MAKE statement.

Initially all three itemvars are "unbound." That is, they are considered to have no item value. Since "p" and "c" are unbound, the element "parent ⊗ c ≡ p" represents an associative search. The LEAP interpreter is instructed to look for triples containing "parent" as their attribute. On finding such a triple, the interpreter assigns the object and value components to "c" and "p" respectively. We continue to the next element "parent ⊗ p ≡ gp." In this element there is only one unbound itemvar, "gp," "p" is not unbound even though it is in the binding list because it was bound by a preceding element. A search is made for triples with "parent" as their attribute and the current binding for "p" as their object. If such a triple is found, its value component is bound to "gp" and the MAKE statement is executed. After execution of the MAKE statement, the LEAP interpreter will "back up" and attempt to find another binding for "gp" and then execute the MAKE statement again. When the interpreter fails to find another binding, it backs up to the preceding element and trys to find other bindings for "p" and "c." Finally when all triples matching the pattern of the first element have been tried, the execution of the FOREACH statement is complete.

In old SAIL, FOREACH elements consisted of either triple searches, set membership, or boolean expressions not dependent on unbound itemvars. Only triple searches and set membership were allowed to bind an unbound itemvar.

New SAIL contains a new way of binding itemvars called a MATCHING procedure. A matching procedure is essentially a boolean procedure which may have zero or more BINDING (written as "?") itemvars as formal parameters. These parameters are not necessarily bound at the time the procedure is called. If the procedure cannot find bindings for its unbound BINDING parameters, it FAILs, causing the LEAP interpreter to back up to the previous element within the associative context of the FOREACH. If it SUCCEEDs, bindings for the unbound parameters will be returned. The matching procedure is actually SPROUTed as a coroutine process. SUCCEED and FAIL are essentially forms of RESUME which return control to the caller with the values TRUE and FALSE, respectively. FAIL also causes the matching procedure process to be TERMINATEd. When the matching procedure is called by "back-up," it is merely RESUMEd. Thus, the entire environment in terms of the procedure's local variables, stack, etc., is the same as when the procedure executed the previous successful return. The matching procedure may continue from the point at which it left off, generating new bindings for its unbound parameters, In many respects matching procedures are similar to the IPL-V "generators" which have appeared in varied forms in other problem-solving languages.

To aid in the binding operations we have provided predicates to determine if a specific parameter is unbound for this call of the procedure. We also have introduced a new form of the FOREACH statement which conditionally adds itemvars to its binding list. Consider the following example of the new form:

```
MATCHING PROCEDURE tool_getter (?
   ITEMVAR tool, tool_type);
BEGIN FOREACH ?tool, ?tool_type | tool ∈
   tool_box ∧ type⊗tool ≡ tool_type DO
   SUCCEED;
FAIL;
END;
```

The binding list of the FOREACH would contain "tool" only if "tool" were unbound. Similarly it would contain "tool_type" if "tool_type" were unbound. The action of the matching procedure is to find a tool if the tool is unknown but the type is known; find the type if the tool is known but the type is not; verify that the tool is of the required type if both are known; or search through the toolbox and return tool, tool_type pairs if neither tool nor type is known. The actual semantics is determined by which, if either, of the parameters are bound.

Unfortunately in general, matching procedures with more than a single potentially unbound parameter are not so easy to code. The user may have to provide up to 2 ↑ N different code sequences to handle the various combinations of N BINDING itemvars.

To illustrate one class of uses of matching procedures let us consider the following problem. We are given a set of cube shaped blocks of varying sizes and are requested to pick a subset of the blocks such that when stacked they will form a tower of a given height. Assume that we will represent a cube by an item whose datum is the height of the cube. We may easily solve this problem by using a recursive procedure "find1."

```
RECURSIVE BOOLEAN PROCEDURE find1
  (SET bset, INTEGER diff; REFERENCE SET
  ans);
BEGIN INTEGER ITEMVAR newb;
  FOREACH newb | newb ∈ bset ∧ (∂(newb)
    ≤ diff) DO IF (∂(newb) = diff) ∨ find1 (bset-
    {newb}, diff-∂(newb), ans)
      THEN BEGIN PUT newb IN answer;
        RETURN (TRUE) END;
  RETURN (FALSE);
END;
```

However, now let us consider a slightly different problem. Suppose we wish to simultaneously build two towers from a single set of blocks. Calling "find1" twice, first with the entire set of blocks for for the first tower, then with the remaining blocks for the second, will not work. Though there may exist many possible subsets which will form the first tower, "find1" will always return the same one even though it is possible to construct the second tower only if a different subset of the blocks were chosen for the first tower. For example, if the set of blocks consisted of sizes 1, 4, and 5 and we were to construct towers of heights 5 and 4, "find1" would construct the first tower using blocks 1 and 4 and thus be unable to construct the second tower.

Now let us see how we would use matching procedures to overcome this problem. Let us write the matching procedure to solve a single tower problem [1],

```
MATCHING PROCEDURE find2 (SET bset;
        INTEGER height; ? SET ITEMVAR ans);
BEGIN
  RECURSIVE PROCEDURE aux
                    (SET s1; INTEGER diff);
  BEGIN INTEGER ITEMVAR newb;
    FOREACH newb | newb ∈ s1 ∧
      (∂(newb) ≤ diff) DO BEGIN PUT newb IN
      ∂(ans);
    IF (∂(newb) = diff) THEN SUCCEED
      ELSE aux (s1-{newb}, diff-∂(newb));
    REMOVE newb FROM ∂(ans);
      END;
  END;
    ans ← NEW({}); COMMENT new item. The
      empty set is datum;
      aux (bset, height);
    FAIL;
END;
```

To call the matching procedure we would simply have a FOREACH statement:

```
FOREACH ans | find2(blockset, height, ans) DO
                  printset(∂(ans));
```

This is clearly equivalent to the solution given above for "find1." However, now consider the two tower case:

```
FOREACH ans1, ans2 | find2 (blockset, height1,
    ans1) ∧ find2 (blockset-∂(ans1), height2,
    ans2) DO
  printsets(∂(ans1), ∂(ans2));
```

This will find a solution if any exists, because if, after finding a solution to the first tower, it is impossible to find a solution to the second problem, we back-up and find a different solution to the first tower and then try the second again.

An interesting distinction between the programs for "find1" and "find2" may be found. Notice that "find1" only returns to its caller after "unwinding" the recursion, thus allowing the answer set to be constructed as the recursion is being "unwound" within a successful call. With "find2," however, the procedure may "return" or succeed while it is still deeply nested in recursion and thus the answer set must be constructed before the next recursive call of "aux" is made.

We envision that matching procedures will be used to simulate n-ary relations, serve as generators of moves or strategies, as well as simply aid in the coding of complex associative contexts.

INTERPROCESS COMMUNICATION

In complicated systems such as the Stanford Hand-Eye system, where there are many cooperating processes present, one would like to have a mechanism by which an occurrence in one process can influence the flow of control in other processes. Such occurrences frequently fall into several basic groups, with perhaps some distinguishing information associated with each occurrence of a given type. In designing interprocess communication facilities for SAIL we wanted to make it easy for the user to distinguish among happenings of the same general type and to define for himself just how each type is to be handled. We have chosen an "event" mechanism which is really a fairly general message processor. Any item may be used as an "event notice," or message, and each type of event in a program is represented by an item. With each such event type, SAIL associates:

1. A "notice queue" of items which have been "caused" for this event type.

2. A "wait queue" of processes which are waiting for an event of this type.
3. Procedures for manipulating the queues.

The two essential actions associated with any event type are

CAUSE(⟨event type⟩, ⟨notice item⟩, ⟨options⟩)

and

INTERROGATE(⟨event type⟩, ⟨options⟩)

where, as elsewhere, ⟨options⟩ may be left out if the default case is desired.

The statement

CAUSE(type1, ntc)

would cause SAIL to look at the wait queue for type1. If the queue is empty, then "ntc" would be put into type1's notice queue. Otherwise, a process would be removed from the wait queue and reactivated, with "ntc" as the awaited item.

If a process executes the statement

itmv←INTERROGATE(type1)

then the first item in the notice queue for type1 would be removed from the queue and assigned to itemvar itmv. If the queue is empty, then itmv would be set to the special item NIC. If a process wants to wait for an event of a given type, it may do so, as in

itmv←INTERROGATE(type1, WAIT)

In this case, if the notice queue is empty, then the process will be suspended and put onto the wait queue for type1.

Similarly,

itmv←INTERROGATE(type1, RETAIN)

causes the event notice to be retained in the notice queue for type1.

This event mechanism should prove useful in problem solving applications in which processes are sprouted to consider different actions. An "or" node in a goal tree, for example, might be represented by

```
:
:
SPROUT (p1, nail (sucevt, boards));
SPROUT (p2, glue (sucevt, boards));
SPROUT (p3, screw (sucevt, boards));
winner←INTERROGATE (sucevt, WAIT);
FOREACH p | p∈{p1, p2, p3} ∧ p≠winner
  DO TERMINATE(p);
:
```

When a branch discovers that it has succeeded, it can execute a statement like

CAUSE (sucevt, THIS_PROCESS);

which would announce success and cause its parent to terminate its less successful brothers.

Events give us a means by which some discovery made by one process can be made to "unstick" some other process which has gotten into trouble. Let's consider our tool getter again:

```
PROCEDURE tool_getter (ITEMVAR
                                      tool_type);
  BEGIN
  ITEMVAR tool;
  FOREACH tool | tool∈toolbox ∧ type⊗tool≡
                                     tool type DO
    RESUME  (CALLER  (THIS_PROCESS),
                                            tool);
  DO tool←INTERROGATE (tool_found,
         WAIT) UNTIL type⊗tool≡tool_type;
  RESUME (CALLER (THIS_PROCESS), tool);
  END;
```

If the FOREACH statement fails to find a tool of the correct type, then tool_getter will be suspended until some process causes an event of type tool_found, using the item representing the tool as the event notice. Suppose that our process "wanderer" has finally gotten a chance to run (everything of higher priority being stuck) and that it does, in fact, stumble across a screwdriver, which it knows to be a kind of tóol. It might then do something like

```
:
MAKE type⊗thing≡screwdriver;
PUT thing IN tool_box;
CAUSE (tool_found, thing, TELL_EVERYONE
                                +DONTSAVE);
:
```

This would cause every process waiting on the event "tool_found" to be awakened. (If no process is waiting, the notice will not be saved on the notice queue.) This would wake up whoever called tool_getter, which would then see if it can use the "thing."

Frequently, one wishes to ask about one of several possible conditions. In some cases this could be done by a simple loop which INTERROGATEs each event type in a list. Unfortunately, if one wishes to wait for an occurrence within a given set of events, this doesn't work very well, since an attempt to wait for one event type will keep the other types from being seen. Therefore, SAIL allows a process to ask about a set or list of event types directly, as in

```
itmv←INTERROGATE

     (ev_type_lis, WAIT+RETAIN)
```

If WAITing is requested, then the process will only wait if all of the notice queues are empty, and it will be reactivated as soon as any of wait queue entries is serviced (All wait queue entries for this request will be deleted.) If it is necessary to know just which type was responsible for a given notice, the option SAY_WHICH may be used. Suppose the statement

```
          itmv←INTERROGATE (ev_type_lis,

WAIT+SAY_WHICH)
```

returns item "notic," which was caused as an event of type catastrophe, as its value. Then the association EVENT_TYPE⊗notic≡catastrophe will be made by the system.

Thus, one way to program an "and" node within process "foo" might be something like

```
SPROUT(p1, fetch(hammer, hand1, sucevt,
                                   failevt));
SPROUT(p2, fetch(nail, hand2, sucevt, failevt));
:
SPROUT(pn, lookat(tv1, boards, sucevt, failevt));
FOR i ← 1 STEP 1 until n DO
  BEGIN
  p←INTERROGATE({{failevt, sucevt}},
                                   WAIT);
  IF EVENT_TYPE⊗p≡failevt THEN
     BEGIN
     MAKE failure_cause⊗foo≡p;
     FOREACH p | p ∈ {{p1, p2, . . . , pn}}
                       DO TERMINATE(p);
     CAUSE(foos_failure_event, foo);
     SUSPEND(foo);
     END;
  END;
CAUSE(foos_success_event, foo);
```

Here, it is assumed that each process is to take responsibility for making "life or death" decisions regarding any subprocesses. As soon as one of the p_i reports failure, foo will terminate all its "children" (whose appointed tasks have become pointless) report its own failure, and suspend itself. If all the p_i report success, then foo will do likewise.

Events may be used together with matching procedures to do deferred updating, as is shown by the following example. A matching procedure may want to make some change to the data base only if the rest of the associative context of the FOREACH succeeds. A simple way of implementing this is to have the matching procedure spawn a process which will do the updating. This process will go into event wait, and the event will only be caused if the entire associative context of the FOREACH succeeds. Consider the following guilt-by-association program. For each member of the suspect list, we first see if he is really undesirable by checking his bank account. If he doesn't have enough money to bribe us we will put another blackmark in the file of anyone who has any association with him, unless that person's only association with his is as an informer (in which case the fink will be given a "negative" black mark). When a person gets 5 black marks he then becomes a suspect.

```
SET badguys; LIST suspect;
MATCHING PROCEDURE linked
                    (BINDING ITEMVAR x);
BEGIN
  PROCEDURE UPDATE;
  BEGIN INTEGER ITEMVAR y, f;
    WHILE TRUE DO
    BEGIN f←INTERROGATE
                         (linkedok, WAIT);
              PUT x IN badguys;
              ∂(f)←∂(f)−2;
              FOREACH y | ANY ⊗ x ≡ y
              DO
              BEGIN ∂(y)←∂(y)+1;
                IF ∂(y) ≥ 5 THEN PUT y IN
                          suspect AFTER ∞;
              END;
    END;
  END;
  ITEMVAR z;
  z←NEW; SPROUT (z, update);
  FOREACH x | x ∈ suspect DO
    SUCCEED;
  TERMINATE (z);
  FAIL;
END;
:
:
:
COMMENT main procedure execution;
FOREACH person, fink | linked (person) ∧
                 (wealth (person) ⟨lots)
  ∧ Informer○person≡fink DO
BEGIN        CAUSE (linkedok, fink);
  :
  :
  :
END;
```

This simple example does of course not really require either matching procedures or the event mechanism to cause the updating, but the technique it illustrates should be quite valuable in more complicated situations.

Although the provided event primitives are sufficient for most of the applications which we have considered, there are some cases for which they are not quite right. For instance, a process might want to wait for a given event only if no other process is already waiting for that event. Instead of trying to provide a special option to cover every possible contingency, we have instead provided a set of queue and process primatives with which the user can write his own CAUSE and INTERROGATE procedures. To substitute his own procedure for the one provided by SAIL, the user makes an association of the form

CAUSE_PROC⊗type1≡new_cause_proc

or

INTERROGATE_PROC⊗type1≡new_int_proc

where type1 is the event type and new_cause_proc and new_int_proc are procedure items bound to the substitute procedures. These procedures will be run as "atomic" operations, and will be allowed to finish without interruption. In particular, any CAUSEs or changes in process status requested by such a procedure will not actually take place until after the procedure exits. This "interrupt level" turns out to be quite useful and permits one to write interrupt handlers that look at a notice of some event, do what they can, and then either just return or else cause an event that will trigger some stronger condition.

CONCLUSION

Each of the features described in this paper was intended to solve particular programming problems. We have not yet had sufficient practical experience with the new system to say with certainty that they are the right ones. There is a great deal of work on these problems in several laboratories and new issues are being raised frequently. We do feel, however, that the basic solutions suggested here will prove useful and that they do significantly extend the capabilities of Algol-like languages.

ACKNOWLEDGMENT

While the work described in this paper was being done, there has also been a significant effort at the Stanford A. I. Lab to produce a new LISP system (LISP 70) which also includes provisions for multiple processes, backtracking, and other similar features. We would like to thank the authors of this effort, Horace Enea, Larry Tesler, and David Smith for several interesting conversations about their system. Although the approach they have taken is somewhat different from ours, these talks provided us with several useful insights.

REFERENCES

1 B ANDERSON
Programming languages for artificial intelligence: The role of non-determinism
School of Artificial Intelligence Univ of Edinburgh Experimental Programming Reports No 25

2 G BIRTWISTLE
Notes on the SIMULA language
Norwegian Computing Centre Publication S-7 April 1969

3 J A DERKSEN
The QA4 primer
SRI Project 8721 Draft Memo 15 June 1972

4 J A FELDMAN P D ROVNER
An Algol-based associative language
Comm ACM 12 8 August 1969 pp 439-449

5 J A FELDMAN R F SPROULL
System support for the Stanford hand-eye system
Proc Second IJCAI September 1971 pp 183-189

6 C HEWITT
Procedural embedding of knowledge in Planner
Proc Second IJCAI September 1971 pp 167-182

7 D V McDERMOTT G J SUSSMAN
The CONNIVER reference manual
MIT AI Memo 259 May 1972

8 E I ORGANICK J G CLEARY
A data structure model of the B 6700 computer system
SIGPLAN Notices 6 2 February 1971 pp 83-145

9 D C SWINEHART R F SPROULL
Sail manual
Stanford Artificial Intelligence Laboratory Operating Note No 52

A survey of languages for stating requirements for computer-based information systems*

by DANIEL TEICHROEW

The University of Michigan
Ann Arbor, Michigan

BUILDING COMPUTER BASED INFORMATION SYSTEMS

Society depends more and more on the recording, analysis, storage, processing, and transmission of data and information. Practically every activity requires an information system. The larger and more organized the activity, the larger and more organized is the information system which serves it. This paper is concerned with Information Processing Systems (IPS) which are built to aid the management and operation of an organization. In particular, the paper is concerned with the methods by which the information needs of the organization can be communicated effectively to those who are asked to implement systems to satisfy the requirements for planning, control, and operations.

The size and complexity of society makes it impractical for a manager or other user personally to satisfy his own information needs, and therefore several functions have evolved with the growth in the use of the computer:

Analysis: Frequently, this term is used with an adjective such as systems, management, or business. The objective of the analysis is to determine, and record, the information needs of the organization and the individuals in it.

Design: The purpose of design is to select the best method of meeting information needs. Since there are usually a number of alternatives available in hardware, software, and processing organization, and since making changes once construction has begun is difficult, it is crucial to design the system as completely as possible before beginning the construction.

Construction: This function consists of building and assembing the modules selected in the design. It includes programming, file construction, hardware acquisition and development of the necessary non-computerized procedures.

In practice, the number of individuals involved in these functions becomes large and some organization is required. One common method is that of a project team which accomplishes all three functions. Another common method is to assign the three functions to separate departments and pass a particular problem from one department to the next, e.g., from analysis to design to construction. (Detailed discussions of the systems building process in use today are available in many papers and books.[1,2,3])

Regardless of whether the project team or functional organization is used, it is of course desirable to document as completely and precisely as possible at each step. The chain of steps of analysis, design, and construction, is only as strong as its weakest link and in practice the chain falls apart first in the lack of adequate documentation from one step to the next.

PRESENT METHODS OF DOCUMENTING REQUIREMENTS

Overview of present methods

The purpose of an IPS, or any group of them, is to serve the organization, and therefore any discussion of the use of the computer must start from the objectives of the organization and the means that its owners and

* This work was supported in part by the ISDOS Research Project, Department of Industrial and Operations Engineering, University of Michigan, and by the U. S. Army under Research Grant DAHC 19-71-G-0005. An earlier version of this paper was published as "Problem Statement Languages in MIS," in E. Grochla (ed.), *Management-Informations-Systeme*, Band 14, Schriftenreihe Betriebswirtschaftliche Beiträge zur Organisation und Automation, pp. 252-282, Betriebswirtschaftlicher Verlag, Wiesbaden, 1971.

managers have chosen to achieve the objectives. As is well-known, it is quite difficult to bridge the gap between the managers and their chosen methods of operating the organization and the precise statements necessary to get computers to do the data processing. There are several major reasons for this difficulty.

First, the organizations are very large and complex and it is not easy for individuals, or groups of individuals, to comprehend all of the interrelationships to the detailed level required for computer processing. Second, the organization has a number of activities going on in parallel and it is difficult to describe everything in a "serial" fashion as is necessary for today's computers. Furthermore, there is no good language for communicating requirements that is understandable by both management and the computer.

This paper concentrates on the techniques by which needs are documented and transferred from the first to the second step, i.e., from analysis to design. The paper is not specifically concerned with the process in the first step, namely, the determination of what the information requirements should be.

Methods for reducing problems associated with documentation of requirements

There have been various attempts to reduce the documentation problem by "shortening" the distance between the person in the organization who needs the information (the user) and the computer. Some methods are listed here in order of the amount of detailed documentation the user must supply, directly or indirectly, from very little to a great deal.

Turning the problem over to another organization

One intuitively appealing approach is for the organization to contract with another for all of its information needs. This has become known as "installation management" or "facilities management." There is not yet enough experience to indicate how successful this will be but in any case, it merely transfers the problem of documentation to another organization. There is certainly more opportunity for this firm to develop expertise in documentation and in fact the absolute necessity of legal, contractual agreements should lead to formal documentation of requirements.

Generalized software packages

In this approach all that is required of the user is to select the package that is appropriate to his needs and to supply the values of the appropriate parameters. Generalized packages[4] are basically of two kinds—application dependent packages and application independent packages. Application dependent packages are generalized programs for performing specific applications such as billing, payroll, accounting, banking and engineering. Application independent packages include report generation and file maintenance, operating system enhancement, simulators and performance monitors, and programming aids. Generalized packages have had only limited success and account for only a small part of the total software development. A recent report[5] estimates the 1972 revenue to be $90 million for applications packages and $110 million for application independent packages. Major interest currently centers on what is probably the most sophisticated example of this approach, the data base management systems,[6] some of which are controlled by parameter values entered on forms or questionnaires—the most widely used example in this category is MARK IV.[6] (Other data base management systems are controlled by task definition and data definition languages.) An example of where user requirements can be stated on forms and directly translated to object code is the Applications Customizer used for the IBM System/3.[7,8]

User-oriented languages

This approach differs from that of generalized software packages in that the user supplies statements rather than parameter values. A user-oriented language is one in which the statements are intuitive and understandable to the user. In the case where the users are managers, the most frequently proposed languages are subsets of English. A number of such languages are in existence but their use appears to be limited to special situations. An example of a user-oriented language intended for management information systems is MUSE.[9]

"Conceptual" frameworks

In these systems the basic framework is provided by the language and is available in a package. This must be supplemented by additional programs unique to the particular situation. An example of this approach is the SIMSCRIPT system for simulation and MAST[10] for business data processing. The user must select the appropriate system and then state his own unique needs usually at the level that permits a program to be written. In practice this approach requires that the user describe his requirements to an analyst or programmer rather than using the language himself.

"Block" system

"Basic Functions" or "Primitives" are defined and usually implemented as macros or sub-routines. The user must then assemble these blocks to satisfy his needs. An example is the BEST system;[11,12,13] several general descriptions exist.[14,15,16] While in theory this approach permits a user to state his needs without a programmer, in practice these systems are used by a programmer. Even for this use, however, these systems to date have received only limited acceptance.

General purpose programming languages (GPPL)

The general purpose programming languages, COBOL, FORTRAN, and PL/1 currently are the most widely used method for building information systems. This category also includes assembly languages which are used whenever optimum use of hardware capabilities is paramount. These languages, of course, require that the user obtain the services of a programmer to implement his information needs.

Relative importance of different approaches

While the above listing has not been supported by quantitative data on relative usage, there are few who would contest the conclusion that by far the largest amount of effort in system building today is based on the use of general purpose programming languages and that undoubtedly this will continue to be true for the foreseeable future. Packages that accomplish "data processing tasks," particularly those now commonly referred to as data base management systems, will come into wider use, and while their use will reduce the amount of programming that would otherwise have to be done, a very large fraction of the total system building will continue to depend on the use of general purpose programming languages. It is therefore worthwhile to examine the system building process based on the use of general purpose programming languages and particularly the inherent problems of communicating between the persons who need the outputs from the system to be constructed and the first automaton in the sequence, namely the compiler. In order to describe these problems it is necessary to make a basic distinction between requirements that an IPS is to satisfy and the processing procedures that will be used to obtain the desired results.

Distinction between information requirements and processing procedures

At the heart of the problem of system building lies the distinction between stating information needs and developing processing procedures that are to be used to satisfy them using the technology of computer-based information processing systems available today. This is a particular instance of the very general concept of goals-means chains. One starts with a goal, lists the various means that could be used to achieve the goal and selects one which then becomes the goal; then the possible means to achieve this goal are listed, one is selected, and so on.

As an example, suppose one is at point A and has a goal of getting to point B. The possible means may be walking, taking a bus, taking a taxi, etc. Assume the taxi method is selected. The goal of getting to point B is communicated to the taxi driver and he selects the means, e.g., the route, etc. Sometimes the passenger will tell the driver the route rather than the destination and sometimes the driver will question the goal (the passenger should go to C instead of B). In general, these actions will be undesirable; in the first case, because the driver presumably knows more about which route is best and in the second because the passenger knows better where he wants to go. This analogy is relevant to the system building situation because ideally the user, or his analyst, should determine the goals of the computer-based system and the system designer and programmer should then select the best method of implementation. All too often, unfortunately, the analyst worries about the best computer means (e.g., the best file structure and record layout) and the programmer worries about the goal (e.g., is this report really needed?). Consequently, both the analyst and the programmer do poor jobs and the resulting system is not effective.

Satisfying the information needs of organization can be represented by a goals-means chain, usually of several stages. The distinction between requirements and procedures at the level immediately before the physical systems design and programming can be illustrated by a simple payroll processing example. (In this simple example, the statement of requirements is represented by one and only one stage. In more realistic examples, several stages of goals-means analysis may be required). The task of the person specifying the needs, i.e., the problem definer, is to describe the requirements for the "target" system which will produce one output: employee paychecks. Certain input information will be available to the target system and the required output type and format is known. These are shown in Figure 1.

In this example it is assumed that the purpose of the

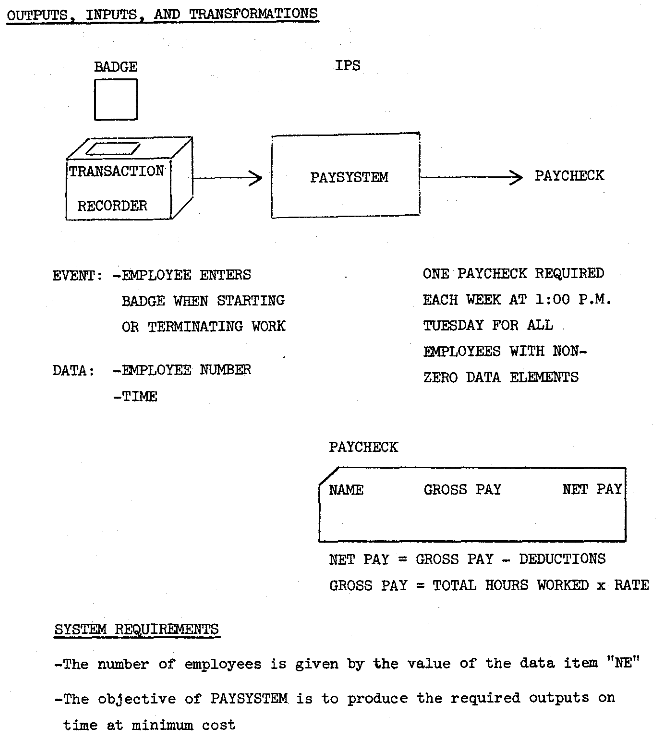

OUTPUTS, INPUTS, AND TRANSFORMATIONS
BADGE
IPS
TRANSACTION RECORDER
PAYSYSTEM
PAYCHECK
EVENT: -EMPLOYEE ENTERS BADGE WHEN STARTING OR TERMINATING WORK
DATA: -EMPLOYEE NUMBER
-TIME
ONE PAYCHECK REQUIRED EACH WEEK AT 1:00 P.M. TUESDAY FOR ALL EMPLOYEES WITH NON-ZERO DATA ELEMENTS
PAYCHECK
NAME
GROSS PAY
NET PAY
NET PAY = GROSS PAY - DEDUCTIONS
GROSS PAY = TOTAL HOURS WORKED x RATE
SYSTEM REQUIREMENTS
-The number of employees is given by the value of the data item "NE"
-The objective of PAYSYSTEM is to produce the required outputs on time at minimum cost

Figure 1—Statement of requirements for the IPS called PAYSYSTEM

target IPS called PAYSYSTEM is to produce one output called PAYCHECK. The time the outputs are to be available is given: each week at 1:00 p.m. on Tuesday for the previous week's work. The number of outputs is specified by saying that one PAYCHECK is required for each employee for whom at least one of the data items included in PAYCHECK other than NAME is different from zero.

The form of the output is stated to be a document containing three data elements: NAME, GROSS PAY, NET PAY. Formulas for computing GROSS PAY and NET PAY are given. PAYSYSTEM must accept one input called EVENT which occurs whenever an employee enters his badge into a transaction recorder. When this occurs the EMPLOYEE NUMBER and TIME are recorded.

Some additional information is given: the number of employees is given by the value of data item NE, and the objective of the IPS is to produce the outputs at minimum cost.

The type of information mentioned above, and shown in Figure 1, is representative of what is necessary to describe the requirements and is sufficient for the purpose of this example, although in a real situation much additional information would have to be specified. For example, Where does the value of NAME come from and how is it associated with the value of EMPLOYEE NUMBER?, How is the value of TOTAL HOURS WORKED determined from TIME?, Where does the value of DEDUCTIONS come from? Additional inputs have to be defined, e.g., to add a new employee to the set of valid EMPLOYEE NUMBERs and to supply changes to the value of RATE and DEDUCTIONS.

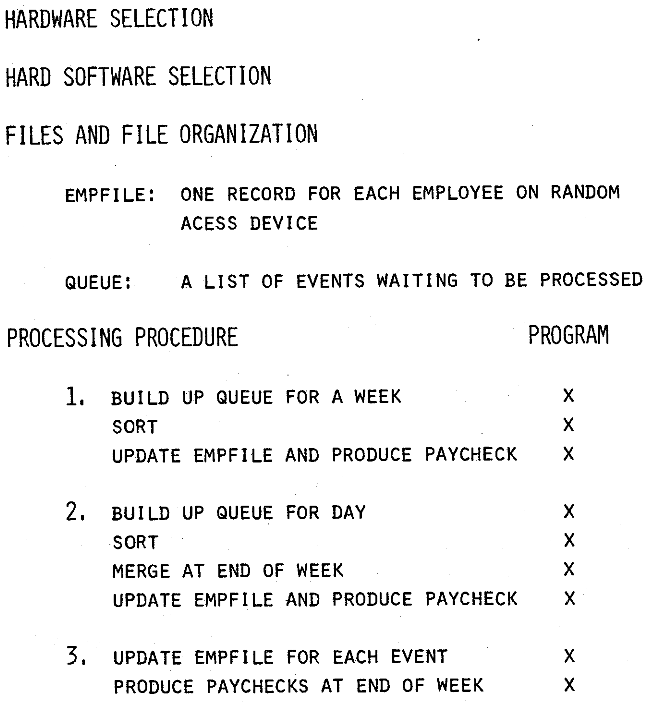
All of this merely corroborates what everyone already knows, namely that stating all the requirements for an organization can be a tedious process. Unfortunately tedium is all too frequently avoided by omitting details that are thought to be obvious and leaving them to the programmer to fill in later.

Under any method of system building the type of information illustrated by Figure 1 has to be collected. Usually this is done manually and is recorded using a natural language (English) augmented by tabular or graphic methods such as decision tables and flow charts. Sometimes an attempt is made to follow the company manual that prescribes standards for documentation.

The deficiencies of manual documentation systems based on the use of natural languages have been analyzed in detail elsewhere[17] and it is sufficient to merely summarize the major shortcomings. While English is a good language for communicating qualitative information, it is too ambiguous for quantitative relationships. The addition of tables, flow charts, decision tables helps a little but major difficulties still remain. The methods are too imprecise, e.g., different data names may be used causing confusion and incorrect results. Manual documentation cannot cope with changes. In large systems the documentation becomes too costly and it absorbs too large a share of total resources. Finally, manual documentation methods based on natural languages cannot be automated efficiently.

In the design process the system designer might follow the procedure outlined in Figure 2. First he will determine the hardware that will be used. Sometimes there is only one alternative, in some other situation this step may require considerable effort. Next he will choose the hard software through which the IPS will be constructed and also through which it will be operated. As one aspect of this process the system designer should consider which of the methods outlined in the previous subsection should be used. In most cases today the method of construction will be through general purpose programming languages perhaps supplemented by data base management systems.

Then the system designer decides what files are needed and what information they should contain. In this case, assume that he has decided that there will be a file called EMPFILE and that it will be stored on a



```
HARDWARE SELECTION

HARD SOFTWARE SELECTION

FILES AND FILE ORGANIZATION

     EMPFILE:  ONE RECORD FOR EACH EMPLOYEE ON RANDOM
               ACESS DEVICE

     QUEUE:    A LIST OF EVENTS WAITING TO BE PROCESSED

PROCESSING PROCEDURE                          PROGRAM

     1. BUILD UP QUEUE FOR A WEEK                X
        SORT                                     X
        UPDATE EMPFILE AND PRODUCE PAYCHECK      X

     2. BUILD UP QUEUE FOR DAY                   X
        SORT                                     X
        MERGE AT END OF WEEK                     X
        UPDATE EMPFILE AND PRODUCE PAYCHECK      X

     3. UPDATE EMPFILE FOR EACH EVENT            X
        PRODUCE PAYCHECKS AT END OF WEEK         X
```

Figure 2—Physical system design

random access unit with one record for each employee and that another file called QUEUE will contain all the events waiting to be processed.

Next, he makes a list of alternative processing procedures (perhaps mentally) and chooses the one which seems to be the best. In this case he might consider letting QUEUE build up for a week (since the output is needed only once a week), sort by EMPLOYEE NUMBER at the end of the week and then update EMPFILE and produce the outputs at the end of the week. In alternative two QUEUE would be built up each day, sorted each day and merged at the end of the week. As alternative three, he might consider updating EMPFILE for each EVENT as it occurred and then producing the output in a weekly run.

The alternative the system designer would choose should be based on the objective stated in requirements. This is frequently a difficult step and may involve much effort if done completely. In this case he may choose alternative three since it is the simplest in concept. However, if he is concerned with processing time he might choose alternative one because it will require less computer time than alternative three. Once the alternative has been chosen the designer then divides the system into parts. Specifications for the various parts of the system are prepared and given to a programmer to write the programs.

The system building process as described above, and as conducted today, depends on manual documentation through the analysis and design phases. Formal documentation begins when the programmer expresses specifications furnished by the system designer in a general purpose programming language. While the basic purpose of this paper is to compare languages for documentation during the analysis phase it is necessary to clearly document why general purpose programming languages are not satisfactory for this purpose, if for no other reason than to dispel the myth, still far too widely believed, that they are.

System building using General Purpose Programming Languages (GPPL's)

While there are methods for causing computers to produce needed output which do not depend directly on GPPL's, it was concluded above that much of the system building in the future will be based on the use of GPPL's or their immediate extensions.

In practice GPPL's are involved in system building only in the construction phase, as shown in Figure 3. The programmer produces source statements which are turned into object code by a compiler. The use of general purpose programming languages causes problems in systems building because by default they frequently are the documentation for the earlier phases. Throughout this discussion COBOL will be used as the example of GPPL since it is now the most widely used in building organizational systems. (The basic arguments, however, are just as valid for the others: FORTRAN, PL/1, and ALGOL, etc.

When COBOL was first developed, it was claimed to have the advantages of being self-documenting and hardware independent. While in most cases it is undoubtedly better to use COBOL than an assembly language, the limitations of COBOL for organizational users of computers are becoming more and more evident: COBOL is a second generation language; it forces the intermixing of business specifications and data processing functions; it results in freezing procedures in the programs; it is not a satisfactory method for communication of information needs; and its use limits the number and size of systems that can be built. The effect of each is discussed further below.

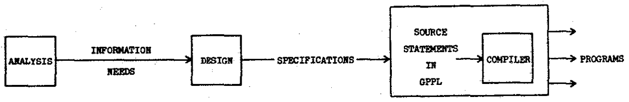


Figure 3—Use of GPPL in systems building

"Second" generation hardware

COBOL was designed to be compiled on second generation hardware. It was developed using experience of another general purpose language, FORTRAN, which was initially designed for first generation hardware; the changes were primarily intended to make COBOL suitable for business data processing, as opposed to numerical calculations. Since COBOL was developed for second generation hardware, it has no facility for dealing with hardware capabilities that are generally available in third generation, but not in second generation, hardware. A program written in COBOL cannot make effective use of random access devices, for example, without some extensions either in the language or in the addition of another language, such as the command language to communicate with operating systems.

COBOL programs are more or less hardware independent of hardware capabilities from one generation to another. The result, however, is performance that is not hardware independent when hardware capabilities change. One immediate consequence of this is that once requirements are implemented in COBOL, the programs must be redone for the next generation, otherwise the result is merely emulation.

As a requirements statement language, COBOL is also limited because much of the information of the type illustrated in Figure 1 cannot be represented. For example, there is no provision for stating that outputs are needed at a certain time or for stating the number of outputs that will be needed.

Intermixing of business specifications and processing procedures

The use of the COBOL language forces the intermixing of the definition of information needs, here called business data specification functions, and the procedures chosen to satisfy the needs, here called data processing functions.

—Business Data Specification Functions (BDSF) define the data manipulation and calculation that must be accomplished to satisfy requirements. Usually these are formulas that define the value for one or more variables, e.g., a BDSF may be "expected sales in a given region, in a given period for a given class of products" or "the total value of inventory at a given time." BDSF are part of the statement of information requirements; in the example in Figure 1, the BDSF are:

NET PAY = GROSS PAY − DEDUCTIONS
GROSS PAY = TOTAL HOURS WORKED*
RATE

In many cases, there may be several ways to define a business function. For example, "inventory value" may be defined using the First In-First Out or First In-Last Out method. It is the responsibility of the user to state the exact definitions he wishes to have used. The BDSF are independent of the particular computer implementation that is used to perform the computation.

—Data Processing Functions (DPF) are the operations that must be used in any particular implementation in order to accomplish the actual computation of the values of the business data specification functions at the time they are needed. For example, in order to (eventually) compute "inventory value," data values such as quantity and price must first be stored somewhere. Then they must be retrieved, multiplied, and summed. Other BDSF may use one or both of quantity and price, and hence, it may be better from a data processing point of view to combine several of these requirements. The DPF used are dependent on the particular hardware and processing procedures selected for their implementation. Data processing functions are selected during the design process; in the example in Figure 2 some of the DPF's used are:

SORT, MERGE, UPDATE

It is essential to be very precise in distinguishing between BDSF and DPF. For example, the user may specify that the IPS in Figure 1 is to produce PAYCHECK alphabetically by NAME. This is different from saying, SORT by NAME. SORT is a DPF which may or may not have to be performed depending on other system design decisions.

The use of COBOL results in a program containing both the BDSF's such as

NET PAY = GROSS PAY − DEDUCTIONS

and the DPF's that the systems designer has selected, e.g.,

SORT

Since both BDSF and DPF are intermixed, usually in relatively complicated ways, it is difficult for programmers to separate out the statements which implement the BDSF from those which implement DPF and it is certainly impossible for a computer program or a user to do so. COBOL designers recognized the necessity to separate data descriptions from the statements in the procedure division. It is now necessary to go one step further and separate BDSF from DPF.

Most organizations are now trying to develop data directories and data bases on an integrated basis for as large a part of the organization as possible in order to avoid duplication and permit comprehensive analysis. In the same way, organizations in the future can be expected to develop directories of BDSF so that they can have standard definitions that can be specified once and used whenever needed.

Freezing processing procedures in programs

One of the consequences of intermixing BDSF and DPF is that the processes are frozen into the program. The programmer expresses the means that the system designer has selected which then become goals to the compiler. The language forces the programmer to specify processing at the level of locating, reading, and writing records and operations on individual data items (PL/1 permits some operations on arrays, i.e., matrices). Therefore, once a program is written units of data cannot be changed without changing the programs. In general, whenever the processing procedures are changed because of changes in hardware, volume of processing, etc., it is necessary to reprogram.

GPPL's as documentation for communicating information needs

When general purpose programming languages were first considered for business problems it was expected that the language being developed, COBOL, could be used for the documentation of information needs. However, this has not happened, as is well stated by Weinberg:[18]

> *"Some years ago, when COBOL was the great white programming hope, one heard much talk of the possibility of executives being able to read programs. With the perspective of time, we can see that this claim was merely intended to attract the funds of executives who hoped to free themselves from bondage to their programmers. Nobody can seriously have believed that executives could read programs."*[18]

COBOL programs are not satisfactory as a communication medium between the user and programmer precisely because they must contain the DPF's. Much of what a user reads when he tries to read a COBOL program is not of interest to him.

Programmer productivity

The amount a programmer can write in COBOL in any given time is limited. Programmer productivity is measured in terms of statements per day—from five to twenty-five. There are not enough programmers to write all the programs that are needed. The reasons for this rate of productivity are partly the difficulties caused by lack of adequate documentation of requirements and partly the fact that the DPFs are programmed many times.

Improvements and extensions to General Purpose Programming Languages

The limitations inherent in the GPPL's listed above are, of course, well-known and a number of attempts to improve or extend COBOL have been made. These need to be listed to examine whether an extended language could eliminate the need for a new requirements language.

"Larger" verbs such as SORT and REPORT WRITER have been embedded in COBOL so that the program is easier to understand and requires less programmer time. Facilities have been added to COBOL to make it possible to use the capabilities of third generation hardware. For example, one manufacturer added IDS to COBOL to make it possible to use random access devices efficiently. Operating Systems and Job Control Languages have been developed to interface the programs and the machines with new capabilities.

These efforts, and the efforts to build data base management systems, are necessary in order to use the present day machines to solve today's problems. However, it is unlikely that such incremental improvements will be sufficient, just as it is doubtful that continued incremental improvement in assembler languages would ever have led to FORTRAN because of the limitations inherent in assembler languages. Similarly, the present effort to solve the problems of adequately documenting information needs by building data base management systems starts by accepting some current features which will, in the long run, limit the effectiveness of the approach.

Need for a requirement statement language

What is required to overcome the difficulties cited above is a formal method of communicating information needs. It must be able to express needs of the type exemplified by Figure 1 without implying any data processing functions of the type selected in the design process exemplified by Figure 2. The analysis in this section has been directed toward showing that general purpose programming languages and their extensions are not suitable for this purpose. The next section will describe a number of languages that have been proposed.

A set of detailed specifications for an ideal "requirements statement language" will then be given.

COMPARISON OF REQUIREMENTS STATEMENT TECHNIQUES

Survey of techniques

The need for a more formal method of documenting requirements for information during the analysis phase has long been recognized. A number of techniques have been proposed. Some of these are listed in Table I.[19–53] Undoubtedly this list is not complete but it includes the known techniques that state as their objective the formalization of statement of requirements or include "analysis" in their title. (Information on omitted techniques is welcome.)

TABLE I—Systems Analysis and Requirements Statement Techniques

Acronym	References	Developer	Status
ADS	19,20	National Cash Register Co.	In use
ASYST	21	Miles Hudson	In development
AUTOSATE	22,23	Rand Corporation	Inactive
CADIS	24,25	J. Bubenko	
CAMIL	26	S. Waters	In development
CASCADE	27,25	Arne Solvberg	In development
CODIL	28	C. F. Reynolds	In development
CORIG		Not known	
DATAFLOW	29,30,31	National Computing Centre	Inactive
IA	32,33	CODASYL Committee	Inactive
	34	H. B. Ladd, W. P. Marcovic	Inactive
LA	35,36,37	B. Langefors	In development
LO	38,39	Lionelle Lombardi	
MINOS		CEGOS	Not known
PSL	40	ISDOS Project	In development
SCOTT	41	SDI Associates	In use
SYMOB	42	Bull, France	Not known
SYNGE		Not known	Not known
SPEC	43	Englic Electric	Inactive
SSP		Robert Brass	In development
SYSTEMATICS	44,45,46,47,48,49	C. B. B. Grindley	In development
TAG	50,51	IBM	In use
	52	Taggart	Not known
UCS		Phillips	In development
YK	53	Young and Kent	Inactive

The basic criteria used for inclusion in Table I were as follows:

i. The language must be designed to state information needs to the system designer and programmer, i.e., it must not permit Data Processing Function statements.
ii. There must be some attempt to develop a formal syntax sufficient to permit analysis by computer programs if desired.
iii. There must be a reasonably detailed description of the language available in the published literature.

These criteria eliminate a number of languages. In particular, all the languages and techniques mentioned in the second section of this paper are not considered further. The second criterion eliminates the (manual) documentation techniques that are part of most system building procedures.[1,2,3] The language developed by Bosak[54] is not included because it is a file processing language rather than a problem statement language. The output decompositions method (ODM)[55] and simulators[56,57] are not included because they are primarily design techniques though they require a statement of requirements in a structured form as input. Programming languages such as APL,[58] Dataless Programming[59] and BCL[60] are eliminated under the first criterion since they require statements describing data processing functions. Software packages concerned with only parts of the information needs such as LEXICON[61] for data definition will be analyzed separately in a later paper.

Previous surveys of some of this literature (and of other related languages) are given by Young,[62] Shaw,[63] Head[7] and in the discussion and proceedings of two workshops.[64,65,25] Shaw's survey includes Information Algebra,[32] Lombardi's Algebraic Data System,[39] Iverson's language,[58] BEST,[11] as well as Decision Tables, IDS, LUCID, ADAM, COLINGO and ATS, which are not included in this paper. Young[62] surveyed BEST,[11] Decision Tables, Lombardi's Algebraic Data System,[39] Iverson's language,[58] Information Algebra,[32] and Young's and Kent's Abstract Formulation.[53] Information Algebra is also discussed by Sammet[66] in the chapter on "Significant Unimplemented Concepts."

All of these techniques have in common the attempt to bridge the communication gap between the Analysis and the Design phases shown in Figure 3. However, a detailed analysis and comparison of all of these proposals is clearly not feasible in this paper. Therefore, a few techniques have been selected for more detailed examination. All these techniques satisfy one or more of the following: they are in current use, they represent areas for further improvement and development or they add to the understanding of the historical development.

Most of the analysis in this section is based on seven selected techniques. The earliest is the work by Young and Kent (YK).[53] Information Algebra (IA) is the result of work by the CODASYL Development Committee.[32] Langefors (LA) published several papers in BIT,[35,36] which have been incorporated into a book.[37] This work is being continued by a number of projects in Scandinavia.[25] Lombardi's Algebraic Data System (LO) was published in COMMUNICATIONS OF THE ACM.[32] Accurately Defined Systems (ADS) was developed by the National Cash Register Company.[19,20] TAG was initially developed by Myers[50] and is described in Reference 51. Grindley has published several papers describing SYSTEMATICS (SY).[44–49]

All these seven approaches are concerned with the problem definition phase of IPS building and hence satisfy the first criterion:

IA: "The goal of this work is to arrive at a proper structure for a machine-independent problem-defining language at the systems level of data processing."

LA: "A formal method for performing systems analysis of information systems in business and elsewhere is needed in order to save systems work and programming and to obtain better systems."

YK: "There are three stages in the application of high speed digital computers to data processing problems:

i. Systems analysis—the task of determining what is to be done.
ii. Programming—a statement of how it is done.
iii. Coding—a translation of this statement into machine language.

This paper presents a first step in the direction of automatic programming as well as a tool which should be useful in systems analysis."

LO: "[The language] relies exclusively on non-procedural representation of processes as sets (tables) of relations between data and results (there are no control statements such as GO TO, etc.) instead of procedure descriptions (which are one-to-one translations of flow charts)."

ADS: ADS is specifically intended for complete specifications of problem requirements: "The completion of ADS forms gives the definer a well-documented application that includes all of the information requirements of the problem."

TAG: "The Time Automated Grid (TAG) technique is a computer tool for use in systems definition, analysis, design and program definition."

SY: "SYSTEMATICS is a language for describing information systems without considering the strategy needed to implement them."

YK, ADS and TAG are problem statement techniques that use a practical, straightforward approach without any attempt to develop a "theory" of data processing. They consist of a systematic way of recording the information that an analyst would gather in any case; any experienced analyst could use either ADS or TAG with very little instruction. IA is more concerned with developing a theory. It uses a terminology and develops a notation which is not at all natural to most analysts. LO is more like a non-procedural programming language than a requirements statement technique, since in order to use it, the system design must be completed, i.e., the file processing runs needed must be known. (The language as described in the literature applies to batch processing only.) The approach, however, is relevant because it presents a "non-procedural" technique for stating processing requirements once the runs are determined. LA starts with a precedence relationship among information sets (files) but does not indicate how these are obtained. This technique therefore is more relevant to the analysis of a problem statement and to the design of a system. However, it does suggest some desirable features of a problem statement technique.

Comparison of features of selected languages

In his review, Young states his difficulty in comparing the techniques he considered:

> "I wish that I could fit all of the developments described here into some sort of nice conceptual framework, but I find it difficult to do so. Each of these systems is intended for a somewhat different purpose, and each implementor has had his own ideas on philosophy and language. Perhaps the best I can do is state what I feel are some of their major advantages and leave as an exercise for the reader any sort of generalization."

These seven approaches, on the surface, appear to be very different but upon detailed examination, they have some major similarities.

All of the techniques take essentially the same view of the problem. The purpose is to describe how to

TABLE II—Comparison of Seven Proposed Problem Statement Techniques

	Information Algebra IA	Young & Kent YK	Lombardi LO	Langefors LA	ADS	TAG	Systematics SY
PROBLEM FORM							
INPUT	INPUT AREAS	INPUT DOCUMENTS	ORDERED INPUT FILES	INITIAL INFORMATION SETS	INPUT	INPUT DOCUMENT AND FILES	not specifically identified
OUTPUT	OUTPUT AREAS	OUTPUT DOCUMENTS	ORDERED OUTPUT FILES	TERMINAL INFORMATION SETS	REPORT	OUTPUT DOCUMENT	
DATA RELATIONSHIPS							
	ENTITY	not used	not used	not used	VARIABLE	DATA NAME	
	PROPERTY	ITEM	FIELD	not used			
	PROPERTY VALUE	not named	not named	not used	VALIDATION RULES		VALUES
	PROPERTY VALUE SET	INFORMATION SET	not used	not used			SETS
	COORDINATE SET						
	DATUM POINT						
	PROPERTY SPACE						
	LINES		RECORDS				
	AREAS		FILES	INFORMATION SET			
	(BUNDLE, GLUMP)						
			BUNDLE				IDENTIFIERS
COMPUTATIONAL RELATIONSHIPS							
	MAPPINGS	PRODUCING RELATIONSHIPS FOR DOCUMENTS	CONTROL PREDICATES	PRECEDENCE RELATIONSHIPS AMONG INFORMATION SETS	LOGIC	PERIOD AND PRIORITY	not mentioned
	COORDINATE DEFINITION	DEFINING RELATIONSHIPS FOR OUTPUT ITEMS	FIELD DECLARATIONS	not used	COMPUTATION LOGIC	not specifically included	DERIVATION RULES
OTHER INFORMATION	none mentioned	VOLUMES ELAPSED TIME	none mentioned	SIZE OF FILES	VOLUMES	VOLUMES PERIOD	none mentioned
PRESENTATION	FREE FORMAT MATHEMATICAL STATEMENTS	GRAPHICAL NOTATION	FREE FORMAT MATHEMATICAL STATEMENTS	PRECEDENCE GRAPH	FIVE FORMS	INPUT/OUTPUT ANALYSIS FORM	TABLES

produce outputs from inputs. All of the techniques provide some method for describing data relationships as the user views them. All provide some way for stating the computational relationships, i.e., the business data specification function. Several provide some method for stating other data such as time and volume. All are concerned with appropriate methods of recording and presentation of requirements.

The techniques are compared in these five areas of similarity and a summary is given in Table II. Each line of this table gives the different names used by different authors. Throughout this section terms that are used with specific meanings in the techniques are capitalized.

Form of the problem

The first category for comparing the seven approaches deals with their view of the overall problem. The following quotes give the authors' definition of data processing systems and their approach to analysis and design.

IA: "An information system deals with objects and events in the real world that are of interest. These real objects and events, called "entities" are represented in the system by data. The data processing system contains information from which the desired outputs can be extracted through processing. Information about a particular entity is in the form of "values" which describe quantitatively or qualitatively a set of attributes or "properties" that have significance in the system. Data processing is the activity of maintaining and processing data to accomplish certain objectives."

LA: "There are some basic propositions made here in connection with the systematic approach advocated, which appear to be in contradiction to present practices or assumptions. One is the hypothesis that in most cases it is possible to isolate and define the relevant organization functions in a separate operation to be performed before the actual design of the system is attempted. It is thus assumed that these functions are defined from the basic goals of the organization and therefore will not need to await the detailed construction of the system. The other hypothesis is that it is possible to define all input information necessary to produce a desired output. The basic assumption here is that actually any information can only be defined in terms of more elementary information, which will then occur as input parameters. Therefore, once a class of information is defined then it is known what input information is required for its production. The point here is that it should not be necessary to work out formulas, or even computer programs, before it can be determined what input data are needed. In fact, it is well possible to work out formulas or programs for an entity, where important variables are missing, so that starting by programming is no safeguard against ignoring important data."

YK: "The content of our analysis is that the objectives of the data processing system have been stated in terms of the required outputs; these outputs are not considered as subject to revision. On the other hand, although the inputs may be organized in any desired fashion, it appears necessary or at least convenient, to state one of the possible input organizations from which any equivalent one can be derived. It should be noted that the input may supply any one of a number of equivalent pieces of information, e.g., either customer's name to be copied directly onto an output or an identification number from which the name can be looked up."

LO: "The common denominator of file processes is the production of output files as functions of input files."

ADS: "The starting point is the definition of the reports—what output information is required. Once the reports are defined, the next step is to find out what information is immediately available. This is followed by laying out the information system in between the output and input. The origin of all information needs to be specified. The outputs of this system are always looked at in terms of inputs."

TAG: "The technique requires initially only output requirements of a present or future system. These requirements are analyzed automatically [by a computer program] and a definition is provided of what inputs are required at the data level."

SY: "SYSTEMATICS is a language solely concerned with techniques and concepts useful to systems analysts in designing information models to meet user's requirements. . . . It is a tool for specifying solutions to information systems problems. More important, it is also a tool for developing such solutions."

Six of the approaches (all except SYSTEMATICS) assume that the problem statement starts at output, e.g., IA: ". . . from which the desired outputs . . .";

YK: ". . . in terms of desired outputs . . ." Therefore, a necessary part of the problem statement should be the description of the desired output. This requirement is implied by LA in the definition of TERMINAL SETS and in YK by the definition of OUTPUT DOCUMENTS. IA does not mention required output as such and, in fact, in the example given in the paper says, "The problem is to create a new pay file from . . ."

TABLE III—Description of Documents

FOR WHOLE DOCUMENT	YOUNG & KENT	TAG	ADS
1. NAME	√*	√	√
2. TYPES OF DOCUMENTS	INPUT,OUTPUT	INPUT OUTPUT FILE	INPUT REPORT HISTORY
3. WHEN PRODUCED	PRODUCING RELATIONSHIP	PERIOD (S,MI, H;D,W,MO,Q,Y) PRIORITY (NUMERIC) FREQUENCY	SELECTION RULES FOR REPORT
4. MEDIA	NOT MENTIONED	NOT MENTIONED	FOR INPUT
5. SEQUENCING CONTROL	NOT MENTIONED	NOT MENTIONED	SEQUENCES
MAJOR			MAJOR
INTERMEDIATE			INTERMEDIATE
MINOR			MINOR
			ARBITRARY NUMBER
6. VOLUME			EXPECTED VOLUME
AVERAGE (A)	√	√	FOR HISTORY AND INPUT
MINIMUM (M)	√	√	AND REPORT
PEAK (P)	√	√	
7. DESIGNED FOR PEAK, AVERAGE OR MINIMUM	NOT MENTIONED	DESIGNER'S CHOICE	NOT MENTIONED
8. OTHER DATA		REFERENCE AUDIT	MEMOS
FOR EACH DATA ITEM	YOUNG & KENT	TAG	ADS
1. NAME	√	√	√
2. HOW USED		FI-FIXED; INFORMATIONAL FF-FIXED; FUNCTIONAL VF-VARIABLE; FACTOR VR-VARIABLE; RESULT	MODIFIED BY — FORMULA — PARTICULAR VARIABLE HOW OFTEN? — NEVER** — PARTICULAR CYCLE** — FIXED TIME** — LOGICAL CONDITION ** by memo only
3. COMPUTATION FORMULAS	DEFINING RELATIONSHIPS FOR VARIABLES ON OUTPUT REPORTS	NOT INCLUDED (MAY BE ADDED AS COMMENTS)	COMPUTATION FORM LOGIC FORM
4. SEQUENCING ROLE	NOT MENTIONED	√	√
5. VALIDATION RULE	NOT MENTIONED	NOT MENTIONED	√
6. FORMAT			
A or N	IN INFORMATION SET TABLE	√	√
SIZE (NO. of CHAR.) FOR OUTPUT		√ Ordering number of P for presence.	√
7. NO. OF TIMES PER DOC. MINIMUM AVERAGE MAXIMUM	√	RATIO	RATIO

* A checkmark denotes provision for including the information listed in the left hand column.

Data relationships

A requirements statement must have some description of the data that will be produced. The most extensive data description facility is the one used by IA. This starts with the concept of an ENTITY which has a connotation of a physical entity in the real world such as an employee, a paycheck, or an order. Each ENTITY has PROPERTIES which describe that entity, e.g., an employee has an employee number, hourly rate, etc. For any given ENTITY there is a VALUE for each PROPERTY. The PROPERTY VALUE SET is the set of all possible VALUES that a PROPERTY can have in the problem. The COORDINATE SET is the list of all PROPERTIES that appear in the problem. A DATUM POINT is a set of values, one for each PROPERTY in the COORDINATE SET, for a particular ENTITY. The PROPERTY SPACE is the set of all DATUM POINTS, i.e., all possible points obtained by taking the cartesian project of all possible PROPERTIES. Once this PROPERTY SPACE has been defined, further definitions deal with subsets of this space. A LINE is a subset which is roughly equivalent to a record and an AREA is a subset roughly equivalent to a file. Other subsets of the PROPERTY SPACE are BUNDLES and GLUMPS. The basic reason for this choice of data description is to use the concepts of a set theory as the formulation for a theory of data processing. (The authors of IA reject data description by arrays as being too limited.)

In YK, the basic units of data are called ITEMS, which corresponds to PROPERTIES in IA. Their term INFORMATION SET is used for the set of all possible values of a particular item and is, therefore, equivalent to the PROPERTY VALUE SET in IA. The information that can be provided for each INFORMATION SET are: (i) the number of possible values, (ii) the number of characters or digits, and (iii) relationships. The following relationships are defined:

	Description	Symbol	Graphic Symbol
Isomorphism	—one to one correspondence		←------→ -------→
Homomorphic	—many to one correspondence	—	
Cartesian product	—P_jXP_k means a pair of P_j and P_k	×	
	—contained in	ε	
Equal to		=	

The relationships may be used to make statements such as, there is one employee number for each employee name and address. YK did not want to make any statements about the file structure and, hence, there are no terms that correspond to records or files. YK also provides a graphical notation for showing relationships.

In LO, the definition of data is more conventional including FIELD (which corresponds to PROPERTY), RECORDS, FILES, etc. The word BUNDLE is used to denote a set of files which is merged on a single input or output unit.

In LA, there is no definition of data corresponding to data items. The problem definition starts with collections of data which are called INFORMATION SETS. This corresponds roughly to the notion of a file in common terminology. LA introduces the concept of an elementary file in which each record contains a data value and enough "keys" to identify it uniquely.

ADS provides three forms on which data is described: REPORT, INPUT, and HISTORY. Each of these forms provides space for some information describing the particular report or input: name, media, volume and sequence and space for each variable. For each variable the forms provide space for name, how the value of the variable is obtained (INPUT, COMPUTATION, HISTORY), a cross-reference, how often the variable appears, and size (number of characters).

TAG provides one form which contains space for data describing the document (or file) and space for each variable. Table III shows a detailed comparison of the data required by YK, TAG and ADS to describe documents and data items.

SYSTEMATICS does not have any rules for specifying structure of data. The major emphasis is on IDENTIFIERS.

Computational relationships—data definition by formula

When general purpose programming languages are used, each program, or sub-program, includes statements which produce output, statements which test the conditions under which the output is produced and statements which compute the values of the variables that appear in the output. It has been argued by Lombardi, in particular, that a "non-procedural" language must separate the statement of *what* output is to be produced when, *from* the statement of the procedure for producing the value of the variables that appear in the output. All seven approaches follow this concept.

In IA the basic operation that the problem definer can use to state his processing requirements is a mapping of one subset of the PROPERTY SPACE into another subset. Two kinds of mappings are defined. One corresponds to operations within a given file. For example, suppose a tape contains time cards, sorted in order by employee number, one for each day of the week. A mapping could be defined which would take the set of (five) POINTS for each employee into one new POINT which would contain the total for the week. The second type of mapping corresponds to the usual file maintenance operation in which POINTS from a number of input files are processed to produce new output files. These two types of mappings are called GLUMPING and BUNDLING respectively. The actual computation of the PROPERTY VALUES of the new POINTS produced by a mapping is specified by a COORDINATE DEFINITION which must contain a computational formula for each PROPERTY in the COORDINATE SET.

In YK the major unit of processing is a PRODUCING RELATIONSHIP; there must be one PRODUCING RELATIONSHIP for each output document. This PRODUCING RELATIONSHIP gives the conditions under which a document will be produced. This statement may contain conditions (Boolean expressions) that depend on values of data ITEM or on time. For example, a PRODUCING RELATIONSHIP might be a "a monthly statement is produced for a customer each month for all customers with a non-zero balance." A PRODUCING RELATIONSHIP may also state that one output Document D_2 is produced for each input Document D_1. The values of the data ITEMS which appear in the output documents are calculated using a DEFINING RELATIONSHIP. There must be one defining relationship for each data item which appears on an output document.

In LO the statements which control whether or not an output record is produced are called CONTROL PREDICATES. There must be one control predicate for each record for each output file. The CONTROL PREDICATES, in general, are Boolean expressions which may involve the use of INDICATORS. The values of the variables which appear on the output records are produced by FIELD DECLARATIONS which are evaluated at the end of each PULSE in a PHASE.

In LA the relationships are given for production of INFORMATION SETS and, hence, correspond to PRODUCING RELATIONSHIPS. However, they are stated only as precedence relationships, e.g., INFORMATION SETS a, b, and c are necessary to produce d. No computational formulas are given. A problem statement may be represented by a graph as shown in Figure 4 where circles represent elementary INFORMATION SETS and rectangles represent PROCESSES.

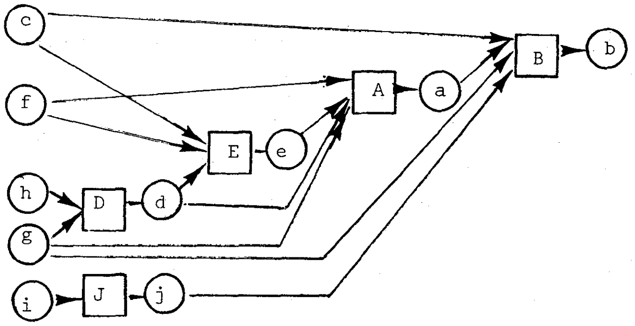


Figure 4—Network representation of problem statement in LA

In ADS some basic information is specified about when reports are to be produced. However, in many cases this is supplied by written notes. This information may be regarded as analogous to the PRODUCING RELATIONSHIPS in YK. ADS requires that each variable be identified as coming from INPUT, COMPUTATION or HISTORY. A form is provided for specifying the computations; this specification is somewhat limited. Another form is used to state logical conditions and these may be used to state when outputs occur and under what conditions computations are performed.

TAG provides for stating how often outputs will be produced by specifying a PERIOD. The available codes are: second, minute, hour, daily, weekly, monthly, quarterly, and yearly. A PRIORITY can be assigned to distinguish a sequence ordering between two documents with the same period. TAG does provide a means for stating which data elements are to be computed but it does not provide for stating the formula for the computation. (The formula can be included in the "Comments" section of the form but it will not be analyzed by the program.)

In SYSTEMATICS there are two types of data items, GIVENS and DERIVED. For each DERIVED item there is a Derivation rule that states the formula by which it is computed. Considerable effort is devoted in SYSTEMATICS to specifying the sets of values over which the rules hold. Consequently data items may be IDENTIFIERS and there are a number of different kinds: PRIMARY, SECONDARY, COMBINED, and COMMON.

Other information

IA and LO do not specify any additional information; LA assumes that the relative size of files is available. YK, ADS and TAG all provide for specifying time and volume requirements.

YK defines two kinds of time: extrinsic (when an event occurs) and intrinsic (the time written on a document). The "operational requirements" consist of a volume for each document (input and output) and a time statement for each output document. Volumes of documents may be expressed in terms of averages over some time period.

ADS permits specification of average and maximum volume in INPUT, REPORT, and HISTORY forms. In addition, each variable in HISTORY is characterized by how long it is to be retained; this may be a fixed number or may depend on a computation.

TAG provides for volume information for documents, size information on data elements and repetition information on data elements within documents. (The information is apparently not used in the programs which process TAG statements.)

Presentation

Both ADS and TAG have well-structured forms for recording the problem statement. They differ in that ADS has five relatively highly structured forms while TAG has a single form. The others do not specify any particular way in which the problem statement must be made. YK describes a graphical method of presentation and LA uses a precedence graph for illustration only.

Analysis, summary and conclusion

Extent of use

Despite extensive recognition of the need for better ways of stating requirements and despite the availability of basic concepts of problem statement languages since 1958 (Young and Kent) and 1962 (Information Algebra) the use in practice of these techniques has not been extensive. Even now ADS and TAG have a limited number of users. Young and Kent's and Lombardi's languages have not been used at all and the development of SYSTEMATICS has not continued after a field trial.[48] Information Algebra has only been used once,[33] and then only for describing requirements for an assembler.

There is no published evaluation of why the techniques are not being used more. There is considerable evidence that many organizations have recognized a need and have attempted to develop their own problem statement techniques but after a while the attempt has usually been abandoned. Comments from a number of such organizations are too subjective to be quoted here, but are incorporated into the following analysis. Specific comments on the use of TAG are that its advantages as a requirement statement language include ease of learning and simplicity in use, its provision of computer processing of requirements data improves ease of modification of the requirement statement, and as a systems design procedure it gives machine-printed copy of program requirements. The disadvantages of TAG as a requirement statement language are that documents cannot be related to each other except through PERIOD, FREQUENCY and PRIORITY and through data elements, that only a two-level data structure is permitted, that repeating groups cannot be handled except through ratios for each variable and formulas cannot be specified. As a systems design procedure, a disadvantage is that it requires manual intervention in the process.

The arguments made against formal problem statement languages can be grouped broadly into two categories: technical problems—the techniques are not satisfactory for stating requirements—and human problems—getting analysts to accept and use them. In practice these two are closely related—an analyst who does not want to change to formal method can usually find some technical reason why the proposed method is not satisfactory. The difficulty here is very similar to that faced in improving other aspects of the system building process.[68,65]

There are a number of reasons that have been suggested for the people problem. One reason is that preparing a rigorous and complete problem statement requires (or at least seems to) more time than the present procedure in which problem statement, systems analysis and programming are collapsed into one indistinguishable process. A second reason is that there has not been any immediate advantage to an analyst or programmer to invest additional time in a more systematic problem statement. Such advantage could come from either or both of the facilities to manipulate the problem statement symbolically and a computer processing of the problem statement itself. TAG currently provides such a software package and one is also available for ADS.[67] (Computer-aided analysis of problem statements will be discussed in a later paper.)

A number of concepts that should be included in a requirements statement language in order to eliminate the technical problem (by ensuring that the formal technique is sufficient to state requirements) are summarized here to provide a basis for the enumeration of desirable objectives of future requirements statement languages.

Form of the problem

There can be no question that the basic purpose of an IPS is to produce outputs. However, it is not clear that limiting the statement of data processing requirements to outputs only (as advocated by TAG) is desirable. Conceptually, one does not want to prejudice the systems design by stating inputs that may not be needed. Frequently, however, certain inputs must be accepted by the IPS and in that case the problem statement might as well include the facility for specifying them. Also the conditions that must be stated (e.g., the PRODUCING RELATIONSHIPS in YK) in the absence of specification of inputs can become very complicated. The statement of problems will probably be simplified if the problem definer can state his requirements either in terms of "events" which require action in the IPS or in terms of outputs required; whichever is most convenient for him, i.e., either in terms of input or of output. Providing this convenience may complicate the analysis of the problem statement by the computer, but the additional processing time is probably worth it.

One objection expressed against viewing IPS as output producers arises from the belief that in the future IPS will be basically data base storage and retrieval systems in which a data administrator will decide what data are to be stored and users will communicate their requests as inquiries. These two views are not incompatible since a result of an inquiry is an output—an output that can be described in the same way as an outout that is produced periodically.

Data description

IA is the only approach that, through the use of the ENTITY concept, attempts to associate data with the real world. It should be noted, however, that the IA language in itself does not depend on how the PROPERTY SPACE is obtained, i.e., whether it is derived from real ENTITIES or from a set of abstract concepts. It is desirable to give the problem definer as much help as possible in defining his data and the analogy to the real world through entities is the best method available. Hence, it might as well be part of a requirement statement language as long as it does not restrict the language in defining data abstractly.

It is important to distinguish between two possible uses of VALUE SETS. (A VALUE SET consists of all possible values of a PROPERTY within PROPERTY SPACE). The first use is for PROPERTIES in which only one value will be in the machine at any one time. For example, the PROPERTY "warehouse number" may have many values in the memory at one time whereas the "quantity" of a particular part number at a particular warehouse will have only a single value at any particular time.

In the first case, the VALUE SETS may be used for validation of input data. ADS, for example, permits validation rules to be given for each data item. In practice, validation is a complex process depending on combination of variables rather than on single variables and such rules are difficult to state on the ADS forms. It may be more desirable to specify validation by defining validation reports as outputs of the system; these then can include any processing specification permitted by the language for specifying data items on output reports.

The second use of VALUE SETS will be in providing information about how much memory space will be required. The basic question is how the problem definer states the role of data items. In COBOL the definition is through the structure definition in the DATA DIVISION and the use of OF and IN; in PL/1 nested qualifiers, separated by periods, are used. In YK the relationships among INFORMATION SETS are used to present this information. In future requirement statement languages it would be desirable to infer as much of the qualifier-identifier relationship of variables from the processing statements themselves and only ask for information that is not included there. It may be possible to obtain all needed information from VALUE SETS and the processing requirements.

The information in a problem statement must be sufficient to infer which data items will have to be stored in the auxiliary memory or in the main memory. A value must be stored in the memory if:

i. It appears in an update statement, e.g., of the form

$$X(, ,) = X(, ,) + Y$$

Here X might be "gross pay to date" and Y "the pay this week."

ii. It is used in a statement without its value having been computed, e.g., "number of exemptions" in a payroll problem. This data item would appear as input on a new hire transaction and in a change transaction and would be used in pay computation.

ADS permits the problem definer to specify data items to be available in HISTORY. These may be either intermediate data items that are used in a number of places or data items whose values the problem definer believes will have to be stored.

It is immediately clear from the preceding paragraph that one cannot determine what data items fall into these categories unless the problem statement contains information about the time at which processing requirements occur. In the first case (i), there must be some way of stating that payroll is computed weekly and the "gross pay to date" is cleared (set to zero) at the end of each year. Similarly in the second case (ii) it must be clear that a new hire transaction only occurs once while the pay computations occur regularly.

Time and volume information

None of the problem statement languages have a well-developed syntax for describing the time aspects of requirements, though YK, ADS and TAG provide some capability. Some help in developing an acceptable "time" language might be obtained by studying the master time routines in simulation languages such as SIMSCRIPT or the executive systems for real-time systems. In a very general sense, time is just one of a number of attributes of data items and, hence, could be included in whatever general data description facility is provided by the language.

It may be noted that time specifications are required not only for determining which data items will be stored but also for determining feasible and optimal storage organizations. The criteria used to determine optimality include both memory space and processing time. One important factor to be considered is organization of data to reduce memory space by such techniques as header-trailer organization as used in hierarchical files and IDS. In order to do this, one must be able to infer the header-trailer relationships from such information as qualifiers and identifiers. Another important factor is the question of what data should be stored semi-permanently and which need only be held temporarily. Again, the analogy to simulation may be useful—SIMSCRIPT, for example, distinguishes between PERMANENT and TEMPORARY ENTITIES.

The second part of the criterion is to reduce processing time. One way this can be done is by reducing the number of accesses to external memory. Since a number of different types of processing requirements must be accomplished, the problem statement must contain both the values of each type and the time periods over which they occur so that accesses to auxiliary memories can be grouped whenever possible.

Both ADS and TAG permit some specification of volume data. Since information about volumes is usually the least accurate part of a problem statement, future languages should have considerably extended capabilities, for example, statement of time and volume by symbolic names.

Presentation

Graphical techniques are extremely useful in many areas of stating specifications, e.g., blueprints for construction specification and flowcharts for algorithms. A graphical technique for the problem statement was given in YK and this has since been extended by Young[69] under the acronym GRIST. The problem statement proposed by LA is equivalent to a directed graph. At the present state of development of problem statement languages it appears unlikely that graphical techniques other than flowcharts and graphs will be very useful. Some experimental work with the proposed techniques including GRIST appears justified, however.

Future problem statement languages will undoubtedly depend on forms, probably somewhere between the two extremes of complete specifications by forms and completely free form. Good forms can be extremely useful in acting as questionnaires and check lists.

Top-down approach

How much information about a problem should be collected when? In current practice the analyst will normally start with the general overall and summary data and gradually he will become more and more specific until he has enough detail to be able to write the programs himself. In contrast, ADS attempts to have the analyst specify all the details of the problem statement at one time.

The best procedure may be compromise between current practice and the ADS approach. Description of data, for example, could be divided into two levels:

Composition—how the data is made up of smaller units of data
Representation—hardware related items such as number of bits, precision, etc.

The composition information clearly is needed as part of the problem statement. Representation information, on the other hand, may not really be needed until program construction begins. A similar categorization could be made for processing requirements. Ideally, a problem statement would require specification of necessary data (composition information) for data, for example, and make optional the statement of information which is not needed until later. This is because sometimes it is easier to record all relevant data at one time.

Mathematical manipulation of the problem statement

IA represents an attempt to develop a problem statement notation that might be manipulated symbolically. The use of set notation and the usual set operations appear a reasonable start for a language in which data processing problems can be expressed. Since to our knowledge IA has only been used once[33] the practical usefulness of IA remains to be demonstrated. It is also not clear how one uses a problem statement expressed in IA in the system design. Both of these questions (the usefulness of IA for problem statement and the derivation of a design from such a statement) provide promising areas for research. Because of the size of data processing requirements it is unlikely that facilities to manipulate the requirements manually will be very helpful. However, there is no reason why such manipulation could not be carried out by computer programs if the language has suitable characteristics.

A REQUIREMENTS STATEMENT LANGUAGE

Objectives of a useful requirements statement language

The discussion in the first two sections has established the need for a better way of stating information needs. The analysis in the previous section has shown that, while there have been attempts to develop such languages, they have not been successful in the sense that they are not in wide use today.

The need for such a language exists even more strongly today and therefore research, development, experimentation and evaluation are needed to develop a satisfactory medium for communicating requirements. A set of objectives for a Requirements Statement Language (RSL) is proposed in this section.

—The language should accommodate the statement of requirements of the kind that are occurring now as well as those that will occur in the future. It is becoming more and more obvious that the cost of changing from one programming language to another is very high. Unfortunately, the present progression from COBOL, to COBOL with extensions, to Data Base Management Systems results in relatively small incremental improvements. The RSL should provide a quantum jump to a completely new generation of capabilities. The characteristics of the situation to be expected in the future that must be accommodated are:

i. Hardware features will increase in quality and reliability. There will be larger hardware with more parallel capabilities—this implies that unnecessary precedence constraints should be avoided whenever possible.
ii. Interrelationship of varying requirements will increase, e.g., jobs with varying priorities, inquiries to be answered, status data to be monitored, outputs required at predetermined times, data to be gathered and results to be distributed over geographically dispersed points, automatic monitoring and control, etc.
iii. The number and type of users with varying interface requirements will increase, e.g., online interaction; data entry such as transaction recorder; interrogation, e.g., reservation clerk, users with no programming needed; system builders; analysts and programmers; data administrators; operators; etc.
iv. Systems will become larger and larger and they will become more integrated. This implies: common data bases, any given programmer does not know what else is going on, new functions such as data administrator, etc.
v. Requirements will be more unstructured; immediate response will be required and requirements will be changing rapidly; jobs require more consistency in data and business data function specifications. This implies that the "user" must be able to communicate with the computer system more directly.
vi. The performance of systems will become more important and hence there will be greater emphasis on more explicit recognition and statement of the criteria by which performance is measured and requirements parameters which affect performance.
vii. There will be more need to monitor the system in operation. The systems change over time either in the volume or the capabilities and consequently there must be provision for changing the internal structure of the system without affecting the correct achievement of the requirements.

—The language should be suitable for use by humans in the necessary activity of determining and stating requirements.

i. The language or part of it must be usable by the manager or his assistants. This is necessary to eliminate the (computer) systems analyst as intermediary in order to reduce the chance for misunderstanding and to reduce the implementation time. To some, this specification implies that the language must be a subset of English. However, the fact that a subset of English is not English can severely limit the value of a subset of English as a requirements language. One of the objections sometimes raised against anything other than a natural language as a requirements language is that a manager will never take the time to use what to him is an unnatural language. It is unlikely that top managers will ever specify detailed requirements. The situation here will be analogous to the current situation in accounting. When a manager first starts out in his career, he is very familiar with the details of accounting and prepares statements for his immediate superior from the reports furnished by the accounting department. As he rises in the organization, he delegates more and more of this to his assistants but he still understands the accounting language and procedures. The career path of the person using the requirements language will be through the management ranks rather than the computer ranks.

ii. The language must be suitable for the top-down approach for problem definition. Most large systems are defined from the top down. The broad, overall outline is developed first and then successively more details are filled in. The language should permit this process and permit checking the problem statement for consistency and unambiguity at each level before proceeding to the succeeding lower levels. The language should, of course, not prohibit the bottom-up approach where this is appropriate.

iii. The language should be suitable for helping in the determination of requirements. It should augment the capabilities of the analysts or teams of analysts who are carrying out the requirements determination.

iv. The language should facilitate the testing and "exercising" of requirements. It is extremely important that statements of requirements be tested before they are implemented. Tests should be made for consistency and completeness. In addition, the person developing the requirements should be able to state data and test conditions that can be used to verify correctness of the requirements statement.

—The language should be suitable for building the system to accomplish the requirements.

i. The language should permit the statement of requirements only and prevent the statement of data processing procedures. This is absolutely necessary in order to make the requirements statement hardware independent and to avoid reconversion costs when the capabilities of the equipment change. It is also necessary to prevent the introduction of restrictions which may limit the efficient use of hardware resources in the later stages of systems building.

ii. The requirements statement must be analyzable by computer programs. The problem statement should not only be readable by a computer program so that the requirements can be stored, but it should also be analyzable so that the problem can be restructured for optimum implementation efficiency without being limited by the sequence used by the problem definers. This is also necessary to permit the automatic construction of the system.

iii. The requirements statement language must permit statement of details necessary for the production of object code. This is necessary if the system is to be constructed automatically. In accordance with the above specifications, however, this detail should not have to be provided all at one time and as much as possible should be available from a library that is built up over time.

iv. The language should permit statements to facilitate the transition process. In most cases, systems already exist with files and programs and it is desirable to be able to move from the present system to the future system in an organized, controlled fashion to reduce inconvenience to the user and reduce cost.

v. The language should be as independent as possible of the particular area of application so that the cost of maintaining separate systems for a number of different applications is eliminated.

Outline of a requirement statement language

A language designed to satisfy the above objectives has been developed and is being tested in the ISDOS Research Project at The University of Michigan, under the acronym PSL (Problem Statement Language). A brief description is given here. A more detailed description is given in the language specifications and user manuals.

The language is used to describe the requirements that refer to the desired target system as a whole, as well as the individual units of the total requirements, i.e., the inputs to and outputs from the target system.

The system requirements include factors such as the parameters that are used in more than one place in the system and whose definition is controlled at the system level; system-wide policies, e.g., the form in which "data" will be used; system constraints that pertain to the system as a whole; resources available to the system, such as hardware, software, etc.; and the performance criterion that is to be used in evaluating the system and in constructing it.

The description of each input to and output from the system contains five major types of information:

i. Identification information which relates the input or output to the external environment. For example, where the input comes from or where the output goes to, who has prepared the statement, what functional area of the firm it is related to. This section will also contain, where necessary, the interface information such as, for example, if the output has to be accepted both on cards and by teletype.
ii. Timing information which describes the triggering of particular input or output in regard to time and/or other conditions. Time here may be specified as real time or time related to some other calendar which would be defined under system requirements.
iii. Volume parameters which determine the quantity of the input or output required.
iv. Data definition showing how the data groups are related by structure.
v. Data definition by formula which gives the individual computations which have to be performed. Decision tables can be used to specify complex logical conditions.

The language will be processed by computer programs where output will be structured to give the analyst as much aid as possible. An overview of the software system is given in Teichroew and Sayani.[40]

CONCLUSION

Since problem statement languages have not been widely used the comparison and analysis in this paper have been based primarily on "paper" systems. The specifications for an ideal requirements statement language have come from this analysis, personal opinion and limited reports from users. There are signs that the situation is changing.

Head forecasts:

> "Most of the work described here is still in its germinative stages, and consequently has had little impact so far on the day-to-day activities of systems people. But it is likely that today's systems analyst, with his still-primitive analytical tools, will one day become as rare as the machine-oriented programmer who flourished a decade ago."[7]

Sammet holds a similar view:

> "Ideally, the user would state only the definition of his problem and the computer system would develop the solution. While the day of asking the computer to "COMPUTE THE PAYROLL FOR MY COMPANY" is at least one or two decades in the future, I believe we will see a large decrease in the amount of detail a user must provide. More specifically, I expect more statements about *what* is to be done and fewer details on *how* to do it. There will be compilers which can effectively determine which of many alternative algorithms should be used in a given situation."[70]

Hopefully, it will be possible to base the next survey of this kind on much more user experience.

REFERENCES

1 R I BENJAMIN
Control of the information system development cycle
Wiley New York 1971

2 W HARTMAN H MATTHES A PROEME
Management information system handbook
McGraw-Hill New York 1968

3 T B GLANS B GRAD D HOLSTEIN
W E MEYERS R N SCHMIDT
Management systems
Holt Rinehart and Winston Inc 340 pp 1968 [Based on IBM's Study Organization Plan (C20-8075-0) 1963]

4 D H SUNDEEN
General purpose software
Datamation January 1968 pp 22-27

5 Computerworld June 28 1972 p 5

6 CODASYL SYSTEMS COMMITTEE
A survey of generalized data base management systems
May 1969 (Available from ACM)

7 R V HEAD
Automated system analysis
Datamation August 15 1971 pp 22-24

8 R T HARRISON
The IBM application customizer services
In 65

9 G W POTTS
Natural language inquiry to an open-ended data library
AFIPS Conference Proc Vol 36 1970 pp 333-342

10 BOEING CORPORATION
MAST: modular application structuring technique
Commercial Airplane Division Seattle Washington no date 58 pp

11 NATIONAL CASH REGISTER COMPANY
BEST manual
1965

12 J R ZIEGLER
A modular approach to business EDP problem solving
Proceedings ACM 20th National Conference 1965
pp 476-484

13 J R ZIEGLER
Computer generated coding
Datamation Vol 10 No 10 October 1964 pp 59-61

14 R T JONES
Basic commercial data processing functions generalized approach to analysis, programming and implementation
IBM Systems Research Institute New York 1964

15 W STIEGER
Survey of basic data processing functions and design using modules
ISDOS Working Paper No 11 August 1968

16 PROCTER & GAMBLE CO
Basic functions manual
Cincinnati Ohio March 1967

17 D TEICHROEW
Computer-aided documentation of user requirements
F Gruenberger ed Information Systems for Management Prentice-Hall Inc Englewood Cliffs New Jersey 07632 1972 pp 97-112

18 G M WEINBERG
The psychology of computer programming
Van Nostrand Reinhold Company 1971 288 pp

19 NATIONAL CASH REGISTER COMPANY
Accurately defined systems
1967

20 H J LYNCH
ADS: a technique in system documentation
Database Vol 1 No 1 Spring 1969 pp 6-18

21 M H HUDSON
A technique for systems analysis and design
Journal of Systems Management Vol 22 No 5 May 1971 pp 14-19

22 O T GATTO
Autosate
Communications of the ACM Vol 7 No 7 July 1964 pp 425-432

23 D D BUTLER O T GATTO
Event-chain flow charting autosate: a new version
The Rand Corporation Santa Monica California October 1965

24 J BUBENKO O KÄLLHAMMER
CADIS: computer-aided design of information systems
In 25 pp 119-149

25 J BUBENKO JR B LANGEFORS A SOLVBERG
Computer-aided information analysis and design
Studentlitteratur Lund 1971 207 pp

26 S WATERS
The CAM (computer assisted methodology) project
Proc of the NCC Workshop on Approaches to Systems Design April 11-13 1972

27 P AANSTAD G SKYLSTAD A SOLVBERG
CASCADE—a computer-based documentation system
In 25 pp 93-118

28 C F REYNOLDS
The importance of flexibility
The Computer Journal Vol 14 No 3 March 1971 pp 217-220

29 M CROWTHER-WATSON
DATAFLOW—a tool for the analyst
3 pp no date

30 R BOOT
Computer-assisted methods in systems analysis
International Conference on Practical Experience in Systems Analyses 18 pp

31 A S LEWIS
File organization for DATAFLOW
File Organization IAG Occasional Publication No 3 Scolts and Zeitlinger N V Amsterdam 1969 pp 165-177

32 CODASYL DEVELOPMENT COMMITTEE
An information algebra phase I report
Communications of the ACM 5 4 April 1962 pp 190-204

33 J KATZ W C McGEE
An experiement in non-procedural programming
Proceedings Fall Joint Computer Conference 1963 pp 1-13

34 H B LADD W P MARKOVIC
Formalized analysis techniques-aids to computer design and computer use
RCA Camden New Jersey 1957 8 pages and illustrations

35 B LANGEFORS
Some approaches to the theory of information systems
BIT 3 1963 pp 229-254

36 B LANGEFORS
Information systems design computations using generalized matrix algebra
BIT 5 1965 pp 96-121

37 B LANGEFORS
Theoretical analysis of information systems
2 Vol Studentlitteratur Lund 1966 (also available from National Computing Centre Ltd Quay House Quay Street Manchester England)

38 L LOMBARDI
Theory of files
Proc Eastern Joint Computer Conference pp 137-141

39 L LOMBARDI
A general business-oriented language based on decision expressions
Communications of the ACM Vol 7 No 2 February 1964 pp 104-111

40 D TEICHROEW H SAYANI
Automation of system building
Datamation August 15 1971 pp 25-30

41 G F RENFER
SCOT simplifies system design changes and time estimates
Canadian Data Systems June 1970

42 D M CAINER
SYMOB (systeme modulaire bull)
Data Processing March-April 1964 pp 106-108

43 ENGLISH ELECTRIC-LEO-MARCONI COMPUTERS LIMITED
SPEC: a technique for expressing system requirements
October 1966 20 pages plus appendices

44 C B B GRINDLEY
SYSTEMATICS—a non-programming language for designing and specifying commercial systems for computers
Computer Journal Vol 9 August 1966 pp 124-128

45 C B B GRINDLEY W G R STEVENS
Principles of the identification of information
File Organization IAG Occasional Publication No 3 Scolts and Zeitlinger N V Amsterdam 1969 pp 60-68

46 C B B GRINDLEY
Specification and interrogation of formal control systems
Paper presented at TIMS XVII Conference London July 1-3 1970

47 C B B GRINDLEY
The use of decision tables within SYSTEMATICS
Computer Journal Vol 11 No 2 August 1968 pp 128-133

48 C B B GRINDLEY
SYSTEMATICS field trials project
31 December 1968

49 P J H KING
Some comments on SYSTEMATICS
Computer Journal Vol 10 pp 116

50 D H MYERS
A time automated technique for the design of information systems
IBM Systems Research Institute New York 1962 50 pp

51 IBM
The time automated grid system (TAG): sales and systems guide
Publication No Y20-0358-1 (no date approx 1968) 40 pp (Reprinted in J F Kelly Computerized Management Information Systems Macmillan 1970 pp 367-400)

52 W M TAGGART JR
A syntactical approach to management information requirements analysis
PhD Thesis University of Pennsylvania 1971

53 J W YOUNG H KENT
Abstract formulation of data processing problems
Journal of Industrial Engineering November-December 1958 pp 471-479 Reprinted in Ideas for Management International Systems-Procedures Association 1959

54 R BOZAK
A proposed file processing language
System Development Corporation Santa Monica California TM 3392/000/00 1967 15 pp

55 M H GROSZ
Systems generation output decomposition method
Standard Oil Company of New Jersey July 1963

56 J W SUTHERLAND
Tackle system selection systematically
Computer Decisions April 1971 pp 14-19

57 D J HERMAN F C IHRER
The use of a computer to evaluate computers
Proceedings AFIPS 1964 SJCC Vol 25 pp 383-395

58 K E IVERSON
A programming language
John Wiley & Sons New York London 1962 286 pp

59 R M BALZER
Dataless programming
The Rand Corporation Santa Monica California July 1967

60 R HENDRY
BCL, a new data processing language
Datamation January 1968

61 ARTHUR ANDERSEN & CO
LEXICON: automation concept for business information systems general description manual
1972

62 J W YOUNG JR
Non-procedural language: a tutorial
7th Annual Tech Meeting South California Chapter ACM March 23 1965 24 pp

63 C J SHAW
Theory, practice, and trend in business programming
AD625-003 Systems Development Corporation July 1965 18 pp

64 R STAMPER
Computer aids in systems analysis
Computer Weekly International August 12 1971 p 18

65 NATIONAL COMPUTING CENTRE
Approaches to systems design
Proceedings of Workshop 11-13 April 1972

66 J E SAMMET
Programming languages: history and future
Communications of the ACM Vol 15 No 7 pp 601-610

67 R THALL
A manual for PSA/ADS: a machined aided approach to analysis of ADS
ISDOS Working Paper No 35 October 1970

68 EDP ANALYZER
COBOL aid packages
Vol 10 No 5 1972 Canning Publications

69 J W YOUNG
Graphical notation for information system description, GRIST
1967

70 J E SAMMET
Programming languages: history and fundamentals
Prentice-Hall Inc 1969

A benchmark study*

by J. C. STRAUSS

Washington University
Saint Louis, Missouri

INTRODUCTION

Several recent articles[1,2] speak to the problems involved in computer system performance evaluation in general and performance prediction for system selection in particular. The interest in this general area is growing in an almost exponential manner as demonstrated by the number of references per year for the period 1965 to 1969 listed in a recent bibliography on Computer Performance Analysis by Miller:[3] 6, 9, 28, 32, 42. However, while there is considerable general discussion in this referenced literature on the relative merits of benchmarking as a technique for system evaluation and selection, there are virtually no documented case studies of benchmarking for third generation, multiprogramming oriented system selection.

This paper partially fills this void by documenting a recent benchmark study conducted by Washington University to aid in the selection of a new, central computer system to replace the current IBM 360/50. The computer-operating systems under consideration include the Burroughs 6714-MCP, IBM 370/155-OS, Univac 1108-EXEC 8, and XDS Σ 9-UTS. The configurations tested are batch processing oriented, constrained to provide certain levels of tape and disk secondary storage and sufficient primary memory to permit satisfactory interactive computing. The principal objective for the new system is to provide significantly increased thruput with equivalent or better tape, disk, and unit record speed and capacity, for the same or less money than the current monthly rent of the IBM 360/50 (~$31,000). It was expected that each of the candidate systems would provide significantly increased capability for interactive computing, better incremental growth capacity, more flexible job scheduling, and no more complex operator and job control capability than that afforded by the 360/50 running under OS. Also of concern was the cost, measured in money, user disruption, and staff energy, of conversion to the new system.

Initially the vendors presented the salient features of their systems (Burroughs—stack architecture and segmentation; IBM—no conversion and RAS [Reliability, Availability, Service]; Univac—proven number crunching and good interactive capability; and XDS—virtual memory and proven time sharing). Washington University, however, had an existing heavy production oriented batch processing load that virtually saturated its 360/50. While there were aspirations to become more involved with terminal oriented computing and the unique architectural features of the new systems were intriguing, it was clear that initial conversion to the new system would be largely mechanical; i.e., make existing production programs work before optimizing efficiency on the new system. Given this emphasis plus a monetary constraint that would not support parallel operation of old and new systems for an indefinite period, it was imperative that any new system must be able to run the existing 360/50 oriented workload with a minimum of conversion. In addition, it was desirable that the new system provide sufficient excess capacity to support new applications and methods development.

The net effect of these various considerations was that despite the vendors insistence that selection decisions be based on their sexy new features, the basic decision had to be strongly influenced by the relative performance of a proposed new system on the existing workload. On this basis it was decided to conduct a benchmark study of the relative performance of various competing systems on a representative sample of the current batch oriented workload. Systems were selected for study based on combinations of past experience, subjective opinions of suitability, and satisfactory relations with local vendor representatives.

The sequel delineates the objectives of the study and presents the design of a comprehensive benchmark to achieve the objectives. The results of the four systems are tabulated and compared to those obtained on the

* This work was supported in part by National Science Foundation Grant No. GJ-33764X.

360/50. Summary comments review the problems encountered in the design of the benchmark and discuss the efficacy of the benchmark study in achieving the objectives. The general description of the benchmark originally presented to the vendors is included in the appendix.

BENCHMARK DESIGN

A working definition of benchmark is developed and the important features of benchmark design are cited.

In Reference 1, Lucas describes a benchmark as follows: "A benchmark is an existing program that is coded in a specific language and executed on the machine being evaluated. A comprehensive series of benchmark runs can demonstrate differences in machine organization and evaluate the performance of I/O equipment and secondary storage...." He indicates that benchmark testing is one of the most effective methods for evaluation of both new system hardware and software for selection purposes. Lucas also makes the following points concerning benchmark series design:

1. The benchmark series should be representative of the current job mix.
2. The proportion of compilation to execution should be considered for each class of programs.
3. All required equipment should be included and usage should be in correct proportion.
4. The priority of various runs (order of presentation) must be in correct proportion.

In addition, Lucas stresses a number of potential problem areas in benchmark testing and provides appropriate references to the literature where those problems have been discussed.

The sequel develops Washington University's approach to benchmark series design against a set of specific objectives.

WASHINGTON UNIVERSITY BENCHMARK STUDY

The objectives of the Washington University benchmark study are established and the design of the benchmarks is reviewed.

Objectives

The principal objective of the benchmark study is to determine the relative thruput of similarly priced and similarly configured computers on a well defined workload. (Table I presents the configurations proposed by the vendors and their approximate monthly cost.) Secondary objectives include determination of:

TABLE I—Proposed System Configuration and Price Comparison

Computer	Configuration	Approximate Monthly Cost
WUCF 360/50	Processor (256K 2μ, 1M 2.5μ) —OS/MFT 484M Bytes 2314, Six Tapes, Unit Record Communications Controller (1-2 Year Lease)	$31,050
B6714	Processor (131K, 1.5μ, 48 Bit Words)—MCP 484M Bytes Disk, Six 144KB Tapes plus Swapping Disk Communications Controller (1 Year Lease)	$28,664
IBM 370/155	Processor (768K)—OS/MFT, MVT 600M Bytes 3330, Six Tapes, Unit Record Communications Controller (1-2 Year Lease)	$39,900
UNIVAC 1108	Processor (196K, 36 Bit Words) —EXEC 8 80M Words Disk, Six Tapes, Unit Record and Drums Communications Controller (5 Year Purchase)	$33,000
XDS Σ 9	Processor (512K)—UTS 600M Bytes 2314, Four Tapes, Unit Record and RAD Communications Controller (5 Year Purchase)	$30,000

1. a measure of conversion difficulty (particularly of the F level COBOL and the sophisticated job control employed by current administrative data processing) based on the conversion effort (man and machine time) required by the various vendors,
2. the extent and usefulness of the standard accounting information collected by the various operating systems,
3. the sensitivity of system performance to tuning of the hardware/software configuration, and
4. relative performance of the systems on internal benchmark characteristics such as: relative CPU times for COBOL and FORTRAN compilation compared to execution, relative CPU time for a series of simple FORTRAN unformatted output jobs that require an alarming amount of CPU time on the 360/50, ease and efficiency of conversion of the six COBOL F jobs to ANS COBOL,

TABLE II—Operational Rules for Benchmark

1. No optimization by optimizing source code (document any changes made)
2. Two basic runs (Run 1 and Run 2) are to be made
3. For Run 1 and Run 2:
   (a) All jobs remain in given classes (if pertinent)
   (b) Degree of multiprogramming is to be less than 5
   (c) Given input ordering of jobs is fixed
   (d) All jobs to be assigned equal external priority
4. Terminate benchmark run when all jobs, but job 7, are complete
5. For Run 1, start execution after all jobs have been read in; record read in, execution and print elapsed time
6. For Run 2, start execution as soon as possible; record total elapsed time
7. Make other optimizing runs as appropriate; record all variances with above rules

> and the capability of the various systems to successfully handle the student oriented jobs currently processed under WATFIV.

The appendix presents the text of the benchmark description that was given to the vendors. Included in this description are detailed run procedures and specifications of expected output. Table II summarizes the operational rules for the benchmark and Table III summarizes the output expected from each vendor.

Benchmark jobs

As indicated in the appendix, the benchmark series consists of 25 jobs: 6 COBOL (IBM COBOL F) compile and execute, 13 FORTRAN (IBM FORTRAN G) compile and execute, and 6 WATFIV jobs. One COBOL (#6) is designed to test compiler diagnostics, but all other jobs in the series are executable. An attempt has been made to distribute benchmark characteristics in rough proportion to those in the actual Washington University workload. Table IV presents a comparison of the distribution of CPU time in the various benchmark tasks and in appropriate tasks in the actual workload. Although CPU time is not the only measure of load representativeness, it is a characteristic measure that is common to all systems in the study. The benchmark series is restricted to three languages (COBOL, FORTRAN, and WATFIV) to facilitate the running of the benchmark on a wide variety of candidate systems. In addition, as discussed in the appendix, the benchmark includes one job the sole function of which is to expend any available CPU time not employed by the other benchmark jobs or the operating system, thereby giving some measure of extra capacity.

TABLE III—Expected Output from Benchmark

1. Description of exact hardware and software configuration
2. All printer and console log output
3. Tabulation of CPU, start and stop time for each job step and job
4. All standard accounting information
5. All necessary source code changes
6. Total man hours and machine time spent initially and by run on benchmark
7. All intermediate results of optimizing runs

As pointed out in Reference 4, representative workload description is extremely difficult for a single multiprogramming operating system; it is virtually impossible across the scope of computer-operating systems under consideration here. From the comparative CPU time figures presented in Table IV, one might question the general representativeness of the benchmark series, particularly in view of the strong emphasis placed on this point by Lucas[1] and others. Based in part on prior experience and in part on the difficulties pointed out in Reference 4, it was felt more important that the behavior of the benchmarks be well understood and cover a broad range of important system features than that the complete benchmark series be representative of the general workload. In this way, as different models of the workload are developed, the benchmark results can be adjusted against the different factors of concern in the perceived model.

TABLE IV—Benchmark and System CPU Usage Characteristics

Quantity	360/50 CPU Benchmark Usage (CPU sec)	360/50 CPU FY72 Second Quarter Usage (CPU min)*
COBOL Compile	447.8	2996
COBOL GO	92.8	657
FORTRAN Compile	82	1459
FORTRAN GO	2311.8	2509
PL/I Compile		1406
PL/I GO		566
Production ADP (Execution of COBOL)		8694
Miscellaneous User Programs		2906
Application Packages		3010
Student Compilers (WATFIV, PL/C)	210.9	1563
Other (Assembler, Link Edit, Utilities, etc.)		3000
	3063.3	28766

* Recording of all job step usage requiring more than .01 minutes of CPU time

Of the five executable COBOL jobs, four are standard Washington University production administrative data processing programs and one is a special job to create an indexed sequential test file for the other four. The COBOL series makes extensive use of disk (both sequential and indexed sequential) and tape files and is representative of a large percentage of the production administrative data processing occurring on the current system. The heavy emphasis on COBOL compilation reflects an expectation that student use will place increasing emphasis on compilation efficiency. It also makes possible some internal benchmarking on the ease and efficiency of conversion of the current 300(+) COBOL F production programs to the ANS COBOL now standard to most vendors.

The 13 FORTRAN jobs are of three types: (1) job 7 is designed to run throughout the duration of the benchmark test and use up any CPU time not employed productively by the other jobs or the operating system; (2) jobs 8, 17, 18, and 19 are heavy compute jobs that test machine precision and the accuracy and precision of standard scientific functions and subroutines; and (3) jobs 9 through 16 constitute an internal check of the efficiency of the output section of the FORTRAN runtime system. Table VIII briefly lists the output formatting characteristics checked by these programs.

The six WATFIV jobs are representative student jobs that had previously been selected from the daily workload to test characteristics of the WATFIV compiler. In the benchmark context, they test the ability of the systems to deal with small, student jobs.

Operational rules

The operational rules are summarized in Table II. Two basic runs are requested to: (1) measure and account for different I/O speeds and control strategies, and (2) determine the ability to conveniently control job input, execution, and output under the different operating systems. The rules listed as 3a-d all relate to attempting to provide a standard, reasonably fair, execution environment across the tremendous architectural breadth of the tested systems. Rule 3a, requiring that jobs retain the class structure (if pertinent) employed on the 360/50 under OS/MFT, is an attempt to ensure that rule 3c relating to retaining the given input ordering of jobs not be violated by IBM when running the benchmark under MFT. Rule 3b placing an upper limit of five on the degree of multiprogramming is an attempt to factor out differences in memory size and allocation strategy on the systems on which the benchmark was run. (It was suggested that those vendors who felt unduly restricted by this rule would make some additional optimizing runs without this restriction.) Rules 3c and 3d are attempts to ensure that *a priori* knowledge of program behavior is not employed to tune the execution of the benchmark to the control strategies of a particular operating system. Rule 4 relates to the use of job 7 to employ all unused CPU time.

Expected output

The output expected from each vendor is summarized in Table III. The results obtained from each of the candidate systems is reviewed in the next section.

BENCHMARK RESULTS

The results of the benchmark series are summarized and aspects of the performance of the individual systems are discussed.

Summary of results

Table V presents the reported vendor effort in machine and manpower hours to convert and run the benchmark series. The Conversion columns list the machine and man hours required to convert the COBOL F to ANS, the JCL, and anything else necessary to achieve a running benchmark. In the case of the 360/50, the conversion time was the time necessary to organize and document the benchmark. The Runs columns of Table V list the time required to run the benchmark series. This time must be interpreted in light of the

TABLE V—Reported Vendor Effort

	Machine Time (Hours)			Manpower (Hours)			
System	Conversion	Runs	Total	Conversion	Runs	Total	No. of Reported Runs
WUCF 360/50	6	1.5	7.5	40	4	44	1
B6714	20	2	22	80	30	110	5
IBM 370/155	20*	60†	80	4	396†	400	30
UNIVAC 1108	5	5	10	160	10	170	2
XDS Σ 9	3	4	7	40	16	56	3(+)

* On 370/145 and including conversion to ANS COBOL which was not required

† Included the generation of several operating systems

TABLE VI—CPU Times (Seconds) of Test Jobs

		WUCF	BURROUGHS	IBM	UNIVAC	XDS
		360/50	6714	370/155	1108	Σ 9
COBOL	1	26.7	26.0	5.2	11.8	25.8
	2	200.2	44.1	45.8	50.4	127.7
	3	50.9	26.5	11.6	19.4	66.6
	4	100.6	36.1	23.8	34.4	70.8
	5	146.8	36.9	35.1	35.3	104.9
	6	15.4	3.9	3.4	4.2	6.0
FORTRAN	7	2000.5	4.2	22.6	196.8	917.3
	8	83.3	47.1	16.9	13.0	30.6
	9	256.7	153.3	61.1	71.5	25.2†(270)
	10	156.1	94.4	39.2	19.7§	28.2†(173)
	11	543.2	632.6	135.0	229.6	24.6†(768)
	12	156.2	131.6	39.2	19.2§	27.0†(163)
	13	135.4	110.9	27.9	67.8	27.0†(186)
	14	33.2	19.0	3.7	34.1§	29.4†(79)
	15	416.1	574.9	96.2	228.2	28.8†(720)
	16	33.3	19.6	3.9	20.0§	28.2†(74)
	17	146.0	107.4	37.6	24.5	35.4
	18	277.9	205.3	75.9	49.5	58.2
	19	74.4	39.4	15.8	11.5	28.2
WATFIV	20	14.7	69.9	8.1	16.9	74.8
	21	6.9	4.5	3.6	3.7	4.2
	22	6.8	1.8	1.4	1.8	1.2
	23	180.4	22.9	12.2	7.9	13.2
	24	0.6	1.3	1.1	1.5	1.2
	25	1.5	1.6	1.2	1.8	1.8
TOTAL *		3063.3	2411.5	704.9	977.7	869.0†(3083.6)

§ Special FORTRAN Library used to block output (reportedly saves 45%).
† Source code modifications to replace WRITE with CALL BUFFOUT.
Approximate CPU times of unmodified programs are included in parentheses.
* Excluding Job 7.

TABLE VII—CPU Times (Seconds) for Compile and Execute of Different Segments

	COBOL	FORTRAN		WATFIV	
		Compute Bound	I/O Check		
System	(#1–#6)	(#8, 17, 18, 19)	(#9–#16)	(#20–#25)	Total
WUCF 360/50	540.6	581.6	1730.2	210.9	3063.3
B6714	173.5	399.2	1736.3*	102	2411.0
IBM 370/155	124.9	146.2	406.2	27.6	704.9
UNIVAC 1108	155.5	98.5	690.1§	33.6	977.7
XDS Σ 9	401.8	152.4	218.4† (2433.0)	96.4	869.0† (3083.6)

* New FORTRAN Compiler is reputed to reduce this by factor of three.
§ Special FORTRAN Library employed.
† Source code modifications; approximate CPU time of unmodified programs is in parentheses.

TABLE VIII—CPU Times (Seconds) of I/O Check Jobs

Program-Characteristics	WUCF 360/50	BURROUGHS 6714	IBM 370/155	UNIVAC 1108	XDS Σ 9
9—Char., Implied DO, A Format	256.7	153.3	61.1	71.5	25.2 (270)
10—Char., Implied DO, Unformatted	156.1	94.4	39.2	19.7§	28.2 (173)
11—Real, Implied DO, E Format	543.2	632.6	135.0	229.6	24.6 (768)
12—Real, Implied DO, Unformatted	156.2	131.6	39.2	19.2§	27.0 (163)
13—Char., Name Only, A Format	135.4	110.9	27.9	67.8	27.0 (186)
14—Char., Name Only, Unformatted	33.2	19.0	3.7	34.1§	29.4 (79)
15—Real, Name Only, E Format	416.1	574.9	96.2	228.2	28.8 (720)
16—Real, Name Only, Unformatted	33.3	19.6	3.9	20.0§	28.2 (74)
TOTAL	1730.2	1736.3*	406.2	690.1§	218.4†(2433)

§ Special FORTRAN Library used to block output (reportedly saves 45%).
† Source code modifications to replace WRITE with CALL BUFFOUT.
Approximate CPU times of unmodified programs are included in parentheses.
* New FORTRAN Compiler is reputed to reduce this by factor of three.

number of runs listed in the last column of Table V that were reported by the various vendors.

Table VI summarizes the CPU times of the individual jobs in the benchmark series when run on the contending systems. Data was obtained on the individual stages of each job, but relatively little new information is present in the detailed results. The single most significant departure of the detailed job step data from observed behavior on the 360/50 is that while the ratio of Burroughs COBOL compile CPU time to that of the 360/50 is .26, the corresponding COBOL execution ratio is .61. Other anomalous behavior is discussed under the performance of individual systems. Table VII presents summarized CPU times for different segments of the benchmark series. Table VIII presents detailed performance on the I/O check portion of the benchmark series. It is interesting to note that all systems appear to have significant problems with explicit use of E Format and A Format as compared to unformatted I/O and with use of Implied DO in the WRITE statement for an Array output as compared to use of Array name only.

Table IX summarizes the thruput performance of the

TABLE IX—Elapsed Times (Minutes) for Execution Portion of Run 1

System	ELAPS (Elapsed Execution Time of Job 7)	Job 7 CPU	CPU′+	ELAPS′†	ELAPS″§
WUCF 360/50	94.9	33.3	51.05	61.6	61.6
B6714*	43.46	0.07*	40.2	43.4	43.4
IBM 370/155*	14.7	0.38*	11.75	14.3	14.3
UNIVAC 1108	21.25	3.55	16.3	17.7	19.1
XDS Σ 9	32.4	15.3	14.5	18.1	63.7
B6724**	25.7	5.2	40.2	25.7	25.7

+ CPU′ is the total CPU time less Job 7 CPU.
† ELAPS′ is ELAPS less Job 7 CPU.
§ ELAPS″ is (ELAPS′) (approximate CPU time of unmodified series)/(CPU′).
* Adjusted priority so Job 7 was active only when nothing else could run.
** The B6724 is a dual processor B6714.

contending systems by presenting the elapsed times for the execution of the benchmark series. Because different vendors chose to interpret the rules somewhat differently, it is necessary to make some corrections on the raw data to arrive at a comparable thruput measure. The column labeled ELAPS is the reported time from start of execution of the first job in the benchmark series (#7) until the last job other than #7 terminates and the operator manually terminates execution. Even if the different vendors had followed the rules and run all jobs at equal external priority it would have been necessary to interpret these results based on the amount of CPU time that job 7 had received. (Job 7 goes through a tight calculational loop and prints out an increasing count every 5000 times through the loop. On the 360/50 there are approximately 6 CPU seconds between counts.) Because job 7 received different relative CPU service due to different internal and external scheduling of the different systems, it is necessary to factor the effect of job 7 from the total elapsed times. An estimate of the upper bound on the elapsed time without job 7 (i.e., with a multiprogramming degree limit of four) is obtained by subtracting the CPU time recorded for job 7 from the elapsed time. This figure is recorded in Table IX in the column labeled ELAPS′. Table IX presents one other attempt to project the elapsed time of the various runs on a comparable basis. The column labeled ELAPS″ presents an estimate of the elapsed time that would have resulted if Univac and XDS had not made modifications to optimize the I/O check portion of the benchmark series based on *a priori* knowledge of benchmark behavior. As indicated in the footnote of Table IX, this estimate is obtained by increasing ELAPS′ by the ratio of the estimated CPU time without modifications to that reported with modifications. Table IX also reports the elapsed time behavior of the benchmark series on a dual processor Burroughs 6700, the B6724. Individual job CPU times are not reported for this configuration because they are identical to those reported for the single processor B6714.

Individual system performance

WUCF 360/50—The Washington University Computing Facilities 360/50 has been carefully configured for good (if not optimum) cost/performance on its broad spectrum administrative, educational, and research computing load. The performance has been significantly improved through replacement of one megabyte of IBM LCS with one megabyte of AMPEX ECM modified locally to run at a 2.5μ sec cycle time. Costs have been significantly decreased by replacing IBM equipment with Potter tapes, Memorex communication controllers, CalComp disks, DATA 100 RJE terminals, and Teletypes. The benchmark jobs were ordered and assigned to classes for MFT scheduling in a way felt to be representative of the standard workload.

Burroughs 6714—Burroughs ran the benchmark series on both their B6714 uniprocessor and their B6724 dual processor systems. Their conversion was good and apparently straightforward. The performance was good especially considering that the benchmark series was strongly FORTRAN oriented and that obviously has not been their forte. Interestingly enough, Burroughs software engineers have since indicated that new FORTRAN I/O now in development will reduce the I/O check jobs (#9-16) from ∼1700 to 700 CPU sec thereby reducing the elapsed time reported in Table IX from ∼43 minutes to 27 minutes or less. Burroughs did not perform Run 1 as specified because MCP did not have (Fall 1971) sufficiently flexible operator control to allow prespooling of the benchmark input. The elapsed time figures presented in Table IX are for Run 2 measured from start to operator termination of job 7 (the first in the series). Burroughs altered the external priority of a number of jobs but with the obvious exception of job 7 this appears to have had little impact on the achieved results. Burroughs also ran several experiments on thruput sensitivity to the degree of multiprogramming and while the results are not clean (external priority was also adjusted between the runs), they

TABLE X—Sensitivity of Elapsed Time

System	Condition : Value	Elapsed Time (Minutes)
B6714	Multiprogramming Degree : 5	43.46
	: 11	43.73
B6724	Multiprogramming Degree : 5	27.58
	: 11	25.73
	: 21	25.81
IBM 370/155	MFT Class/Partition : Fixed	14.7
	: Dynamic	14.1
XDS Σ 9	FORTRAN Compiler : FORTRAN	18.1
	: FLAG	21.3

are interesting; these results are included in Table X. It is gratifying to note that the dynamic segmentation of the B6700 provides real insensitivity of thruput to the degree of multiprogramming.

IBM 370/155—IBM converted and ran the benchmark on their 370/155. The 155 performance is certainly the best of any of the systems tested, but the cost is definitely the highest as well. Initial conversion work done on the 370/145 which is in the target price range indicates that the 145 would have performed well, but it was not tested because it did not allow sufficient primary memory to provide the desired interactive environment. The relatively large times reported in the Runs columns of Table V were due in part to the large number of optimizing runs and in part to the generation of several specially tuned operating systems. IBM tried many combinations of scheduling strategies, memory sizes, and operating systems types. The general impression from all their presented data is that OS is relatively insensitive to the performance tuning that they reported. Table X presents a representative sensitivity comparison. For a fixed relationship between job class and memory partition, the elapsed time was 14.7 minutes. Based on *a priori* knowledge of job behavior, the class/partition relationship was changed during the benchmark series; the resulting elapsed time was 14.1 minutes.

Univac 1108—As indicated in Table IX, the 1108 performed very well. Table VII establishes that 1108 performance on the compute bound FORTRAN jobs was the best of those tested and performance on the COBOL jobs was credible considering that the benchmark series was run on a system using slower FASTRAND II drums in place of disks. Unfortunately, as might be expected from 36 bit word orientation of the 1108 and the drum orientation of the EXEC 8 file system, the conversion effort indicated in Table V is relatively high as compared to other more IBM oriented vendors.

XDS Σ 9—Initial reports on the Σ 9 performance were amazingly good. Tables VII and IX indicate that before knowing of the implications of their source code modifications, the XDS overall cost/performance was better than any other tested system. Later measurements, however, established that XDS suffered even more severely than the other vendors on the I/O check portion of the benchmark. Use of the FLAG (FORTRAN LOAD AND GO) compiler in place of the standard FORTRAN compiler led to the increase in elapsed time reported in Table X.

CONCLUSIONS

There are two obvious and important conclusions to be drawn from this study.

1. That the general benchmarks described here could be run with relatively little effort on such a wide diversity of machines speaks well for the standardization of COBOL, FORTRAN, and general operational procedures.
2. When establishing rules for benchmark operation, it is imperative that the vendors fully understand the meaning and importance of each constraint.

Of natural interest is the question of which system did Washington University select. The answer is none, which in light of today's troubled economic picture is probably not surprising. The systems that the University could afford on a monthly cost basis require too large a conversion cost and the one system with minimal conversion cost (IBM 370/155) is too expensive for the needs of the University. In addition, while in the process of conducting and analyzing results of the benchmark, it became obvious that the concept of a single central computer had to be carefully reviewed in light of the rapidly developing technology currently surfacing in the small machine area. These observations probably lead to an obvious conclusion/warning for vendors running benchmarks; i.e., qualify the budgetary and political position of the prospective customer at the highest management levels.

REFERENCES

1 H C LUCAS JR
Performance evaluation and monitoring
Computing Surveys Vol 3 No 3 1971

2 M E DRUMMOND JR
A perspective on system performance evaluation
IMB Systems Journal Vol 9 No 4 1969

3 E F MILLER JR
Bibliography on techniques of computer performance analysis
General Research Corporation Report P O Box 3587 Santa Barbara California 93105

4 M D DRAPER R C MILTON
Univac 1108 evaluation plan
Technical Report No 13 University of Wisconsin Computing Center Madison Wisconsin March 1970

APPENDIX—BENCHMARK DESCRIPTION GIVEN TO VENDORS*

Washington University Computing Facilities Benchmark September 10, 1971

Purpose

This document briefly describes the composition, run procedures, and expected output from a test batch

* The appendices referred to in the text of this appendix are not included.

stream of 25 jobs, hereafter referred to as the WUCF Benchmark.

Composition

The WUCF Benchmark consists of 6 COBOL jobs, 13 FORTRAN jobs, and 6 WATFIV jobs. APPENDIX A describes the 6 COBOL jobs named BMJOBl through BMJOB6. APPENDIX B describes the 13 FORTRAN jobs named TSTJOBO7 through TSTJOB19 and the 6 WATFIV jobs named JOB20 through JOB25. APPENDIX C lists the order that these jobs appear on the job input tape marked JOBSTR which is card image 9 trk, 800 bpi, unlabeled, unblocked, and LRECL=80. In addition to the job input tape, the WUCF Benchmark requires two data tapes: (1) the gifts file tape marked 000763 which is 9 trk, 800 bpi, IBM standard label (VOL=SER=000763), one logical file, data set name of GIFTTAPE, DCB parameters: RECFM=F, BLKSIZE=290; (2) the development master tape marked 000720 which is 9 trk, 800 bpi, IBM standard label (VOL=SER=000720), one logical file, data set name of DRAWOFFD.EV,DCB parameters: RECFM=F, BLKSIZE=370.

The printer output of the WUCF Benchmark when run on the WUCF 360/50 is attached as APPENDIX D and a copy of the corresponding console log is attached as APPENDIX E.

The first job, TSTJOBO7, on the job input tape is somewhat special in the following sense. This job is intended to employ any CPU time not used by other jobs. In this way, the CPU time not used due to our restrictions on job classes and multiprogramming degree may be assessed. Due to the scheduling behavior of the Dynamic Execution Monitor employed by WUCF, this job actually employs some usable CPU time. Since for Run 1 and Run 2, you are restricted from changing relative job execution priorities this will almost assuredly be the case for your Run 1 and Run 2 as well.

Run procedure

The WUCF Benchmark is to be run at least twice (Run 1 and Run 2) on your proposed configuration. If you employ a job class scheduling system, the jobs are to retain the same class groupings as now. The degree of multiprogramming is to be no greater than five. For these first two runs (Run 1 and Run 2), the present input ordering of the jobs on the job input tape (APPENDIX C) is to be preserved and all jobs are to be assigned equal external priority.

Run 1 will involve reading all jobs into the input queue(s) prior to execution, starting execution, terminating execution when all jobs but TSTJOBO7 have finished execution (terminating TSTJOBO7 from the console!), and recording elapsed batch reading time, execution time, and print out time. (The attached output from WUCF 360/50 included in APPENDICES D and E was run in this manner.) Run 2 will involve starting execution and read in simultaneously, terminating execution when all jobs but TSTJOBO7 have finished execution (terminating TSTJOBO7 from the console!), and recording the elapsed time from start of reading of the first job to termination of print out of the last job.

Expected output

The WUCF Benchmark documentation is to include the following for Run 1 and Run 2:

1. description of exact hardware and software configuration,
2. all printer and console log output,
3. tabulation of CPU, and start and stop times for each job step and job, and
4. all accounting information accumulated by your standard operating system for each job step and job (e.g., records in, lines out, CPU time, disk space, channel times, device times, etc.).

In addition, you may choose to optimize the bench mark timings for the proposed system through the use of different orderings, priorities, job classes, degree of multiprogramming, etc. If you do, please label these results as Run 3, Run 4, etc., and describe the alterations performed. If for any reason, the configuration listed in 1 (above) is any different than that proposed, please describe the reason and the expected performance differences on the proposed system. Please do not optimize the benchmark by optimizing the source code of the test jobs. All necessary source code changes must be documented.

In order that we may be able to assess the difficulty of optimizing throughput, please document total man hours and machine time spent initially and by run on the benchmark. In addition, we should like to see all intermediate results of optimizing runs so that we can determine system throughput sensitivity.

Toward an inclusive information network

by R. R. HENCH and D. F. FOSTER

General Electric Company
Bethesda, Maryland

INTRODUCTION

In the next decade, an increasing proportion of computer power will be provided by information service networks, rather than by the tens of thousands of individual installations which now exist. This proposition is made attractive by many economies of scale, of which "Grosch's Law" is only one. Equally important are:

- The reduced cost of the redundancy which is required for high reliability.
- The drastically reduced cost of nationwide communications when provided in large quantities.
- Lower operations costs.
- Better capacity utilization due to more incremental growth.
- Variable, rather than fixed, costs for the user.

General Electric is committed to providing a viable service alternative to in-house processing for a wide variety of computing requirements. This paper describes the design of the systems currently under development which addresses that requirement.

DESIGN ALTERNATIVES

In developing a service for the entire DP community, there is an almost irresistible temptation to say, "Let's start from scratch and do it right." Under this philosophy (which we will call the *exclusive* alternative), the designer would develop a new and highly-generalized combination of hardware and software. Such a system aims at being all things to all men—it serves timesharing and remote batch; the on-line user and the massive tape sort; the big LP problem and the inventory-control system. Unfortunately, this alternative ignores the existent multi-billion-dollar investment in software and knowledge for current systems. It also ignores the advantages which may follow from specialized hardware and software dedicated to specific kinds of applications or functions.

The other alternative (which we will refer to as the *inclusive* alternative) is to take advantage of pre-existing systems, integrating them into a Network but still maintaining each as an entity. Thus, advantage may be taken of the software and knowledge which exist for such systems, in addition to taking advantage of their unique capabilities.

The key problem, of course, is in providing meaningful integration among these systems. Many major application areas involve the interaction of diverse kinds of computing resources. For example, data may be collected on-line from geographically-distributed locations, then submitted for batch processing. If the online and batch functions are to be done on separate systems, a high degree of integration must clearly be provided between them.

DESIGN STRATEGY

GE has therefore undertaken the development of what we may refer to as a *Technology of Compatibility*, the aim of which is to permit the integration of diverse kinds of systems. The specific elements of this strategy are as follows (See Figure 1):

- Provide a generalized, widely deployed *Communications Network.*
- Provide an *On-line Service* of the highest possible quality and efficiency.
- Interface this foreground service with a variety of existent *Background Systems*, both GE-owned and (potentially) customer-owned.

Thus, the user will be provided with a common set of tools for on-line program development and for geo-

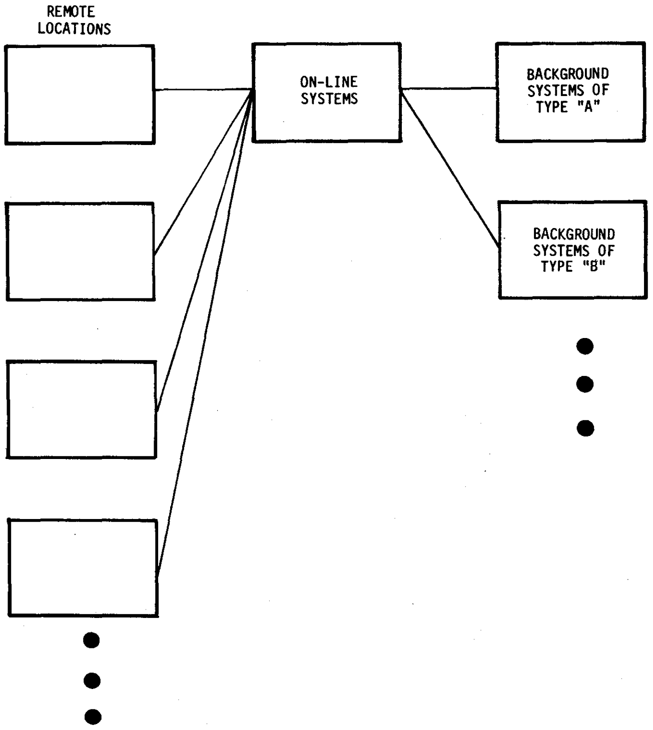

REMOTE LOCATIONS
ON-LINE SYSTEMS
BACKGROUND SYSTEMS OF TYPE "A"
BACKGROUND SYSTEMS OF TYPE "B"

Figure 1—Conceptual design

graphically-dispersed data collection and distribution. At the same time, he will be provided with a choice of background systems, depending on his specific needs. Ongoing batch applications may thus continue to operate in a compatible environment while being provided with an efficient "front-end" for interactive processing.

The rest of this paper will be devoted to a discussion in some detail of the elements of this strategy—the communications network, the online service, and the foreground-background interface.

COMMUNICATIONS NETWORK

The Network extends to over 250 cities in North America and Europe. From each of these locations, the user may reach any of the central systems by typing an appropriate user number. The Network design permits the systems to be located physically anywhere within the Network.

Topologically, the Network is a polycentric star (Figure 2). The transmission and distribution paths throughout the entire Network are store and forward two-line logic with diversified routing wherever necessary. The Network is based on the use of distributed computers performing specialized functions.

Distribution—remote concentrators (Honeywell 416)

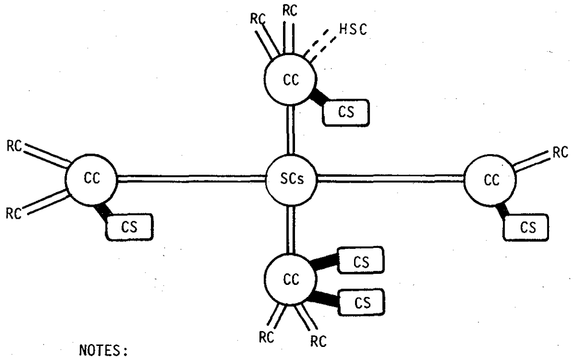
These computers are the outermost nodes of the grid. Their functions include:

- maintaining awareness of physical terminal characteristics (speed, character set, etc.)
- simple editing (line and character deletion)
- context recognition. RCs keep track of whether each terminal is entering *data* or *commands.* Thus, the central system will be interrupted only when commands (which require immediate central system action) are entered, and not every time the terminal user types a line of data.

Transmission—central concentrators (Gepac-4020)

CCs provide the interface between the central systems and the Network, and also between the RCs and the Network. Their functions include:

- buffering data to and from the terminals. A drum is provided for this purpose.
- determining to which central system a user is to be connected, at logon time.



NOTES:

SC/CC LINKS ARE 4800-14400 bps.

CC/RC LINKS ARE 2400 bps.

CC/HSC LINKS ARE 9600 bps.

FREQUENCY DIVISION MULTIPLEXERS ARE USED BEYOND THE RC LEVEL.

Figure 2—Network structure

- communicating with central systems, remote contrators, and switching centers.
- maintaining awareness of Network configuration.

Switching—switching centers (Gepac-4020)

The switching centers provide a transparent link between Central Concentrators. At present there are two SCs, operating independently of each other logically as well as physically.

Processing—central processors (GE635, 605, 235, HISI 6000)

Central systems provide all user job processing. A number of central systems may be attached to each CC. The MARK IIs, accounting for most of the load, are connected directly via memory-to-memory interfaces. Any other type of central system may be connected with a modified interface (either hard-wired or remote) which uses a high-speed communications channel on the CC and a standard software interface discipline.

Network bandwidth extension

The Network has recently been extended to support synchronous terminals up to 4800 bits per second. This has been done consistent with the Network structure through the introduction of the high-speed concentrators (Diginet 1600s) interfacing with CCs much as remote concentrators do now. Message size and queuing philosophy are the only differences. HSC-CC communication will initially be via dual lines at 9600 bps. The HSCs will support a variety of terminal types, including computers. An OS teleprocessing package has been developed to permit 360 or 370 computers to communicate with the Network. Through a procedure called Interprocessing,® a user may transmit files between his in-house system and one of the Network's on-line systems. Thus, he may achieve nationwide access to his data bases for inquiry, updating, and further processing.

THE ON-LINE SERVICE (MARK II)

MARK II was designed and optimized specifically for interactive use. This includes conventional timesharing, but more importantly the very different and demanding area of geographically-distributed, transaction-oriented processing. The hardware (GE-635) provides a master/slave concept and automatic hardware relocation of programs. The software provides the following general capabilities:

- a simple, user-oriented command system.
- task string logic. Each command is broken down into a string of standardized, elemental tasks. Tasks may schedule other tasks ahead of or behind themselves.
- heavy use of resident, reentrant code.
- a separate communications processor.
- a completely device-independent logical file system.
- multichannel swapping, permitting up to three swaps to be going on simultaneously.
- extensive security and integrity checking.

In addition, a number of capabilities oriented specifically toward the transaction-oriented data collection and distribution market have been provided. These include:

- file permissions, by means of which each user may exercise explicit control over access to his data base.
- file locking, permitting control over multiple-update situations.
- journalization, providing a common magnetic tape for the logging activities of multiple users.
- interprocess communication, whereby programs can signal other programs indicating processing tasks for them to perform.
- user control. The activities of a terminal user can be placed under complete program control. When the user logs on, a specified program may be automatically invoked, and escape may be prohibited. Almost all system functions may be done by means of CALLS from a high-level language program, eliminating the necessity for the eventual terminal user to give system commands.

THE BACKGROUND INTERFACE—MARK III

As discussed earlier, it is our intention to interface MARK II with a variety of background systems. The first step in this direction is MARK III, which provides an interface between MARK II and the GECOS-III remote-batch system. Although GECOS runs on hardware similar to MARK II, it is a completely different operating system, optimized for batch and remote job entry to the same extent that MARK II is optimized for on-line activity. The interface permits each system to specialize in the kind of work at which it is best:

- all on-line work is handled by MARK II
- all background work is done by GECOS
- separate file systems are maintained
- a high-bandwidth interface is provided. All background work is submitted by MARK II across the interface.

Design principles

Although the specific code used to interface MARK II with GECOS-III is probably not transferable in interfacing MARK II with other systems, the fundamental design concepts most certainly are. These are as follows:

- Changes to the MARK II and GECOS-III operating systems must be kept to a minimum. Absolutely no changes could be made which would in any way impact current users of MARK II or which endanger compatibility with future manufacturers' releases of GECOS.
- The foreground and background systems must be independent of one another. The failure or intentional shutdown of one must not affect the other.
- The interface or mailbox protocol must be adaptable to communications technology and not involve physical constraints such as distance or storage.
- It must be possible to interface multiple MARK II systems to a single GECOS-III system.
- It must be possible to transfer very large files and output reports between the systems efficiently.
- Character and file type conversions must be handled as straightforwardly and as automatically as possible.
- It must be possible to use the background system conveniently from either a high-speed or low-speed terminal.

Physical interface

The initial linkage between the MARK II and GECOS-III systems is a shared disc device (Figure 3). This is *not* in any sense a shared file system; it is simply used as a high-bandwidth communications channel. The transmission of messages between the two systems is completely asynchronous, using what is called a "passive mailbox scheme." One disc mailbox area is written only by MARK II and read by GECOS-III. Another area plays the converse role. All interface status information is maintained on the disc itself, to facilitate restart in the event of the failure of one of the systems. It should be understood that the use of the disc is merely a practical solution and plays no active role in the logical interface discipline.

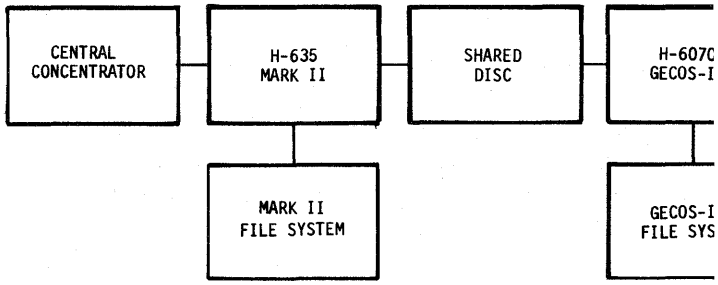


Figure 3—MARK III

Logical interface

A background job may originate from either a high-speed or a low-speed terminal. In either case, the request will be generated on MARK II and will be transmitted across the interface to GECOS. Background job initiation is via the BACK command. BACK requires a set of *directives* describing both the interface work and the background work to be done. At present these directives include:

- run a job file, i.e., place it in the background system input stream.
- transfer given files between foreground and background, with appropriate file format and character set conversions.
- create and purge background files.

Standard system job output (SYSOUT) is automatically returned to the foreground system and placed in a special library. At the user's option it may be:

- directed to a high-speed terminal.
- scanned by a low-speed terminal, using special editing features.
- placed in the user's permanent file system.

Conversion disciplines

A variety of character sets coexist in MARK III. ASCII is primarily used by the foreground system and by low-speed terminals; BCD is used by the background system. Both ASCII and EBCDIC are used for communicating with high-speed terminals. This proliferation of character sets is a fact of life for anyone who aspires to serve today's and tomorrow's information society. In order to avoid chaos in dealing with it, it was necessary to establish a system-wide discipline for dealing with conversion problems.

All files stored on MARK II have their character set content (when meaningful) retained in the catalog. Processes which must perform conversions will utilize this information. Conversion processes will retain identity of logical records, and will also maintain special control information, such as printer slew control characters. The conversion is thus considerably more complicated than a simple one-for-one transliteration. By default, all data files transmitted are converted from the natural conventions of the host machine to those of the target machine. SYSOUT files are, however, maintained in original format and with the same record structure which exists on the background system. This is done for several reasons:

- to avoid unnecessary conversion. Much background output is simply scanned, using the low-speed terminal editing procedures; then thrown away.
- to save file system space (6 bit BCD characters as opposed to 9 bit internal ASCII representation).
- to avoid double conversion. If output is eventually destined to a high-speed terminal, for example, its character set will probably be EBCDIC. The user, however, may well not decide whether or not he wishes to print his output on a high-speed terminal until he has scanned it.

Software implementation

- BATCHER, a MARK II slave software module, is invoked whenever the BACKGROUND command is given. It checks validity of the directives and queues the job for transmission.
- INPUT MONITOR is a continuously-running slave module which takes jobs from the queue and writes them to the shared disc. At this point all files which are to be taken over to the background are merged in.
- MARK2 is a GECOS privileged-slave module which handles all communications with MARK II via the shared disc. It is written in reentrant code and can communicate with multiple MARK II systems simultaneously.
- PREP (preprocessor) is a job spawned for each user ahead of his requested background work. It handles conversions, file creations, etc.
- OUTPUT MONITOR is a MARK II continuously-running slave module which handles all output returning from GECOS across the interface. A variety of other status and editing modules are also provided.

It is significant to note that the actual changes to the background system have been minimal. This validates the design philosophy of truly connecting alien operating systems without compromising software maintenance by the hardware vendor.

Applications programs

GE has always made heavy use of applications programs to provide higher-level interfaces to its systems. This policy will be continued with MARK III. Several kinds of application programs which operate across the interface are being developed:

- control card generators. These programs will interactively query a user and will generate for him the control cards necessary to perform most basic background functions.
- application initiators. These will exist in connection with a specific background application program (for example, linear programming). In addition to generating the control cards, they will perform validity checking on the user's input data and spawn the optimum job relative to resources needed.
- distributed applications. These are application packages which make use of both the foreground and the background, applying each to the tasks it is best at. For example, an accounts receivable package might maintain customer credit level information in the foreground, but keep customer name and address records, purchase records, etc., on tape in the background.

THE FUTURE

GE's developments in the Information Services Business have been based very largely on distributed intelligence techniques. Future work will largely consist of extensions of these concepts. Some significant future developments are discussed below.

Multiple background systems

As indicated earlier, the evolution of the MARK III concept to various and diverse background systems is planned. As more systems are integrated into the total service, there will follow closely the natural extension of remotely-located backgrounds. With time the interface will permit background systems owned and operated by customers to become an integral part of the product. The implications are obviously very penetrating. It will be possible to procure a highly reliable interactive foreground service capable of meeting all the demands of continuous on-line services, which in turn is coupled with one's own in-house machine. From the same terminal one could cause activities to be spawned back into one's own system.

Shared second-level file system

In any multicomputer configuration it becomes increasingly undesirable to require manual intervention for handling of such removable media as tapes. For this reason, we are closely following the evolution of massive storage device technology. When this technology becomes sufficiently mature, we will implement such a device, shared among all systems in a given location and provided with intelligent controllers. This will create a hierarchical storage structure in which the lower level of the hierarchy is common to many systems. Information will move from one level to another at the explicit or implicit request of users.

Communication of complex data structures

Initially, file transfer across the interface has been limited to character-oriented sequential files, since these are common to all operating systems. It is desirable to provide transfer of more complex file types, including random structures and binary data. Clearly the problems involved in converting such files from the conventions of one operating system and set of hardware to those of another are non-trivial, but it is possible that a reasonable solution can be obtained for some substantial subset of all files.

Multiple foreground systems

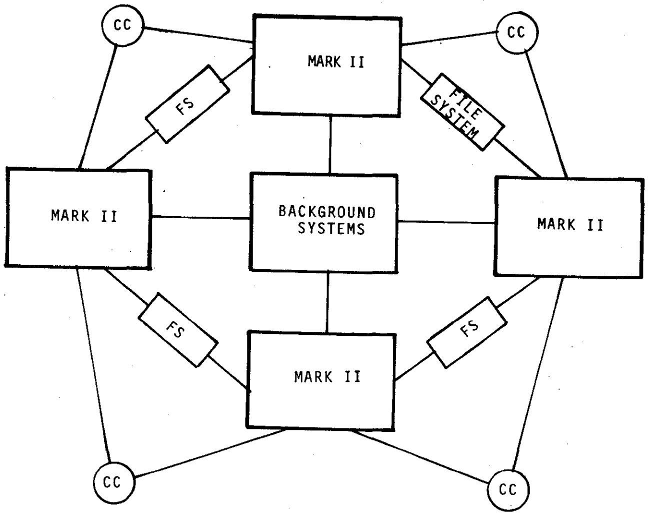


Figure 4—Ring system

At present, MARK II provides a high degree of reliability through the use of a redundant "swing system". If a system fails, its users may be transferred to the swing system by logic in the communications processors. Peripheral switches move files logically from any seriously crippled system to an idling reserve. This does not address the interruption problems of on-line services but has made a tremendous stride in availability.

The next major step to extend the foreground reliability will be changing the current processor focus to a file focus. The current architecture is so structured that the system topology is a function of processors. This must change to make the pivotal nodes the file subsystems. This will be done by having the file and processor subsystem interlaced in a concentric circle as shown in Figure 4. Users will be assigned to central systems dynamically by the communications processors. If a system fails or is taken off-line, its load will immediately be taken up by the two adjacent systems. In a short time the load of the failed system will be distributed evenly across the ring. A stable ring of n systems reduced to $n-1$ will re-stabilize within 30 minutes due to normal log-ons and log-offs.

In addition to its reliability advantages, this configuration provides significant economies by balancing load fluctuations across systems. Preliminary analysis indicates that these economies may exceed 15 percent of total capacity.

COMMENTS

The Network described here has grown in a somewhat organic manner, subject at all times to the hard discipline of commercial viability. It is not our intent to lay down an inflexible master plan, but rather to develop a general structure with the ability to adapt to changing conditions. Inevitably, some aspects will be developed more intensively than others as experience indicates their value. The character and scope of those features which are commercially implemented are therefore subject to alteration. Hopefully, the structure described will provide sufficient flexibility to accommodate this continuing redesign and to provide for the smooth integration of new technologies into the ongoing operation.

ACKNOWLEDGMENT

The ongoing developments discussed here are implemented under very tight schedules and by a surprisingly small group of people. The authors would like to express our appreciation to our colleagues since this has been a group effort from its inception.

REFERENCES

1 *On line time share computer services—A telecommunications industry survey*
TELECOMMUNICATIONS Volume Four Number Eleven November 1970

2 G E GAINES JR J M TAPLIN
The emergence of national networks—Remote computing—Year VI
TELECOMMUNICATIONS Volume Five Number 12 December 1972

3 R C HAAVIND
The three phases of shared computing
COMPUTER DECISIONS November 1971

4 G J FEENEY
Information network technology
General Electric Company Document No 238010 2/70 Presented at the Congres International d'Informatique Paris September 23 1969

5 J C CASTLE
Communication system design in multiple access computer networks
Memorias de la Conferencia Internacional IEEE
Mexico 1971 Sobre Sistemas Redes y Computadoras
January 19–21 1971 pp 636-640

6 F D MONTGOMERY JR
Computer resource sharing through communications network technology
EUROCON 71 Digest IEEE Catalog No 71 C 56—REG 8 October 18-22 1971 Lausanne Switzerland

7 W S HOBGOOD
Evaluation of an interactive-batch system network
IBM Systems Journal Volume II No 1 1972

8 L G ROBERTS B D WESSLER
Computer network development to achieve resource sharing
SJCC 1970

Computer jobs through training—A final project report

by M. GRANGER MORGAN, NORMAN J. DOWN and ROBERT W. SADLER

University of California at San Diego
La Jolla, California

INTRODUCTION

Two years ago we presented a preliminary report on a computer-science training program for disadvantaged high school students and young adults which we had developed at the University of California at San Diego.[1] At that time we had completed the construction of our mobile classroom facility and had tested our special hands-on curriculum on several small pilot classes, but we had not completed any full job-training courses and had only limited experience upon which to base conclusions and recommendations about the effectiveness of such training.

Since early 1970, the Computer Jobs Through Training project has offered in-depth counseling and referrals to roughly 730 disadvantaged young people, excluding those who have entered our training programs. We have enrolled roughly 330 students in pre-vocational, in-school, and other training programs and, in addition, have graduated 24 young adults from extended full vocational courses. This paper provides a summary of what we have learned.

PROGRAM OBJECTIVES AND COURSE PERFORMANCE

Our original work in the field of computers and the disadvantaged began in the summer of 1968, when we ran a course for a group of high school students. We found that the disadvantaged student was somewhat less disadvantaged in this field than in others because success depended more upon innate logical abilities and less upon social and cultural prerequisites than was true in most other fields. On the basis of this initial experience, we decided to attempt the development of a general training program for San Diego area students. Because the disadvantaged population in San Diego is widely dispersed, and because we felt that there are important psychological advantages to bringing the training into the students' own community, we undertook the development of a mobile computer classroom in a 40-foot truck trailer. We initially conceived of the program as designed for in-school use at the high school and junior high school levels, but as we became increasingly aware of the urgent need for job training in San Diego and the difficulties of working with the public schools, we began to think more in terms of vocational training for young adults who were without college plans. By the time that we had actually begun the serious development of our physical facilities, we had become heavily committed to a vocational program.

Our curriculum was developed to involve student hands-on participation beginning on the first day. It is heavily problem oriented, with emphasis on the logical and algorithmic aspects of the work. An extensive set of 35 mm slides was developed which support the main course and also offer background on peripherals and hardware operation, case studies of many different types of computer applications and systems, and support in areas such as basic math and business world interpersonal relations. Finally, a collection of lab materials and demonstrations were developed in digital logic and switching circuit theory. A description of our facilities and curriculum is available in our previous report.[1] Figure 1 shows the classroom interior.

Figure 2 is a summary of the courses we have operated since early 1970. We began with a single vocational course in San Ysidro, which is principally a Chicano, or Mexican-American, community. Our performance in that course is summarized in Figure 3.

Because this program was severely underfunded, we were unable to begin further vocational courses. In order that the hardware be used at a reasonable level, we began to talk extensively with people in the San Diego Unified School District. After having learned the ropes, we found that working with the public schools was not the serious problem we had imagined provided one is willing to let the system proceed in its own way and at its own pace. Under contract with the District's

Figure 1—Students in the latter stages of a vocational course work at debugging a COBOL applications program in the Computer Jobs Through Training mobile computer classroom

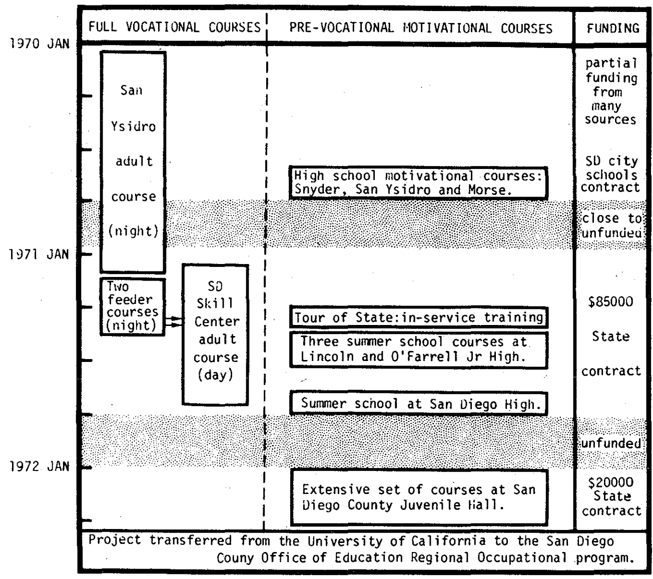

FULL VOCATIONAL COURSES
PRE-VOCATIONAL MOTIVATIONAL COURSES
FUNDING
1970 JAN
1971 JAN
1972 JAN
San Ysidro adult course (night)
Two feeder courses (night)
SD Skill Center adult course (day)
High school motivational courses: Snyder, San Ysidro and Morse.
Tour of State: in-service training
Three summer school courses at Lincoln and O'Farrell Jr High.
Summer school at San Diego High.
Extensive set of courses at San Diego County Juvenile Hall.
partial funding from many sources
SD city schools contract
close to unfunded
$85000 State contract
unfunded
$20000 State contract
Project transferred from the University of California to the San Diego Couny Office of Education Regional Occupational program.

Figure 2—A summary of the course activities conducted by the Computer Jobs Through Training project during the period from 1970 to 1972. Frequent uncertainties and discontinuities in the funding situation resulted in severe restrictions on class recruitment and scheduling, particularly in the case of the longer vocational programs

compensatory education program, arrangements were made to run three summer school programs, two at Lincoln Senior High School and one for entering seventh graders at O'Farrell Junior High School. Both we and the school system were very impressed by the results of these courses, particularly with respect to student motivation. We began once again to seriously think about making such in-school programs a major part of our activities.

During the summer and fall of 1970, we began to sense a shift in the data-processing employment market in San Diego. We had previously held extensive conversations with employers in late 1969 before we undertook our first vocational course and therefore knew that appropriate jobs existed. However, as the recession began to settle in in earnest, we became seriously concerned about the problem of placements, especially for future courses. As unemployment levels began to rise in San Diego (see Figure 4), the amount of training and experience which defined the "entry level" programmer climbed rapidly. We soon felt that major emphasis in our future adult courses should be directed toward preparing students for further study at the junior college, college, and university levels rather than toward direct job entry. We argued that it was unreasonable for us to duplicate existing training programs but that instead we should concentrate on the difficult

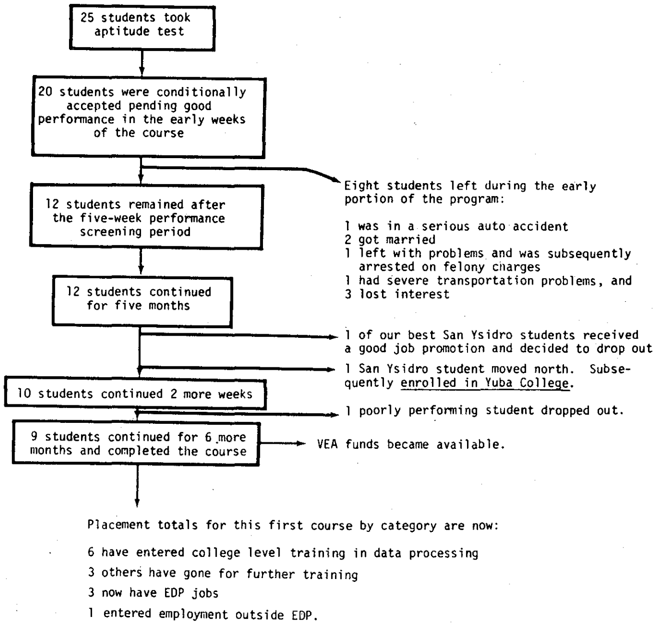

25 students took aptitude test
20 students were conditionally accepted pending good performance in the early weeks of the course
12 students remained after the five-week performance screening period
12 students continued for five months
10 students continued 2 more weeks
9 students continued for 6 more months and completed the course
Eight students left during the early portion of the program:
1 was in a serious auto accident
2 got married
1 left with problems and was subsequently arrested on felony charges
1 had severe transportation problems, and
3 lost interest
1 of our best San Ysidro students received a good job promotion and decided to drop out
1 San Ysidro student moved north. Subsequently enrolled in Yuba College.
1 poorly performing student dropped out.
VEA funds became available.
Placement totals for this first course by category are now:
6 have entered college level training in data processing
3 others have gone for further training
3 now have EDP jobs
1 entered employment outside EDP.

Figure 3—A description of performance for the first adult vocational course run in San Ysidro on the southern edge of San Diego, just north of Tijuana

task of motivation during the early portion of the training and then, once we had students "hooked," rely on the more conventional college-level programs to complete the job.

In late 1970, we received an $85,000 contract from the California State Office of Vocational Education (under the Federal Vocational Educational Act PL90-576 Part A section 102b). In requesting this support, we proposed a multilevel program which would include straight vocational courses, heavy emphasis on training for further education with close coordination between our project and the local junior colleges, and a substantially expanded in-school pre-vocational and motivational program for high school students. Essentially all of the programs shown in Figure 2 during 1971 were funded under this contract. Figure 5 shows approximate performance figures for these courses.

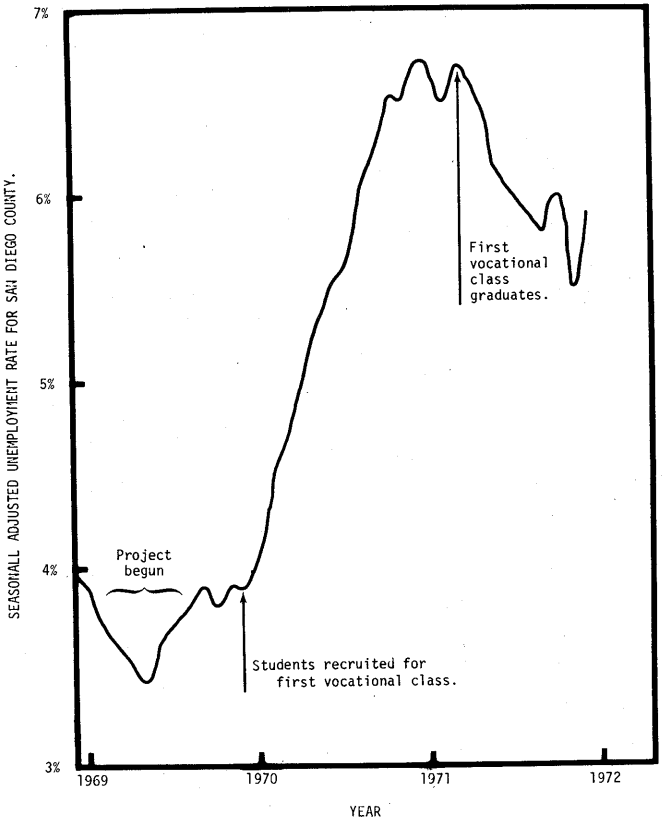


Figure 4—Seasonally adjusted unemployment rates for San Diego for the period from 1969 to 1972.[2] Note the enormous change in the employment situation which occurred between the time when the project was developed and the time when the first course was graduated. High unemployment in San Diego hit particularly hard in the lower-level aerospace jobs and among the disadvantaged, the two population groups with which the CJTT project was concerned

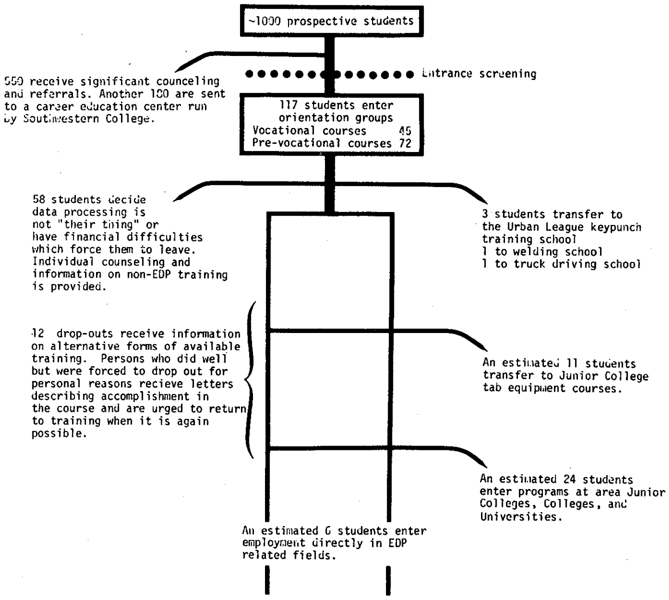

~1000 prospective students
Entrance screening
550 receive significant counceling and referrals. Another 100 are sent to a career education center run by Southwestern College.
117 students enter orientation groups
Vocational courses 45
Pre-vocational courses 72
58 students decide data processing is not "their thing" or have financial difficulties which force them to leave. Individual counseling and information on non-EDP training is provided.
3 students transfer to the Urban League keypunch training school
1 to welding school
1 to truck driving school
12 drop-outs receive information on alternative forms of available training. Persons who did well but were forced to drop out for personal reasons recieve letters describing accomplishment in the course and are urged to return to training when it is again possible.
An estimated 11 students transfer to Junior College tab equipment courses.
An estimated 24 students enter programs at area Junior Colleges, Colleges, and Universities.
An estimated 6 students enter employment directly in EDP related fields.

Figure 5—Approximate performance figures for courses run during 1971 under State-administered VEA support. Because of difficulties in funding continuity, proper follow-up activity was not possible for most of these students and hence many of the above numbers are estimates

One of the principal lessons we have learned is the misleading, or at best incomplete, nature of numerical performance criteria. We have trained a substantial number of people, but the intuitive feeling that remains is that our greatest impact has frequently not been in the area of formal training but rather in shaping the world views and attitudes of our students. The Computer Jobs Through Training staff has come from many different backgrounds, both culturally and ethnically. In mid-1971, we were three Anglos, nine Blacks, and four Chicanos. Nevertheless, the one thing that we held in common was a high level of professionalism. Nine of the seventeen employees at that time were experienced programmers, and all had a very real interest in the problems and welfare of our students. This combination of professionalism and concern has been the principal reason for the kind of impact that we have been able to have on our students. Few of those who entered our courses had developed any feeling for what the professional does, how he acts, what he thinks, or why. When they left, most of our students had some real understanding of the professional world, an enhanced image of themselves and their capabilities, and a greater ability to cope with the system of the dominant culture. We know of no way in which to adequately measure such effects without spending more on measurement than we spent on the program itself, and yet all of the senior staff agree that it was in this area that we had our greatest impact. Even those students who have not pursued further work or study in computer science after leaving our program have gone away with a different view of themselves and their world—a view which we feel significantly enhances their potential for upward mobility.

COMMUNITY AND STUDENT ENVIRONMENT

San Diego's disadvantaged community is geographically dispersed. The climate is balmy and the political atmosphere relaxed. As a result, there have been no major political upheavals to coalesce this community into positive and forceful action. The local offices of the national organizations that minister to the needs of the disadvantaged are, in many cases, staffed by an executive officer imported from another city and worried principally about his own career, and an operational staff of middle-aged, middle-class people whose attitudes fail to reflect the political developments of the past ten years. This fact, together with local politics, and competition for limited federal funding and the community limelight, have left many of these organizations unresponsive to the needs of their community.

Our project has been crucially dependent upon the minority undergraduate members of our staff to carry a heavy portion of the teaching load. And yet, this same problem that characterizes the San Diego disadvantaged community as a whole characterizes these students. Their contribution and dedication is great, but they have not shown the aggressive leadership that similar university students might have shown in other larger cities such as New York and Los Angeles, where the disadvantaged communities are more politically developed.

Although San Diego has a number of agencies that counsel the poor and sometimes refer them to appropriate training and jobs, it has very few programs that actually offer training for the disadvantaged. In some cases, because of the lack of crosstalk between these agencies, people get referred from one referral agency to another several times in a row. Our experience, as one of the programs offering training, is that outside counselors showed little understanding of what we were offering and, in many cases, lacked discretion in their referrals to our program. Our recognition of this problem led us to develop our own counseling staff and a booklet designed for student use that included details of the various training and social-service programs offered in San Diego, information on how to apply, who to see, and directions including bus routes to each facility. Every person who came into contact with us was given a copy of this booklet as part of our counseling process and most received individually written followup letters from our counseling staff.

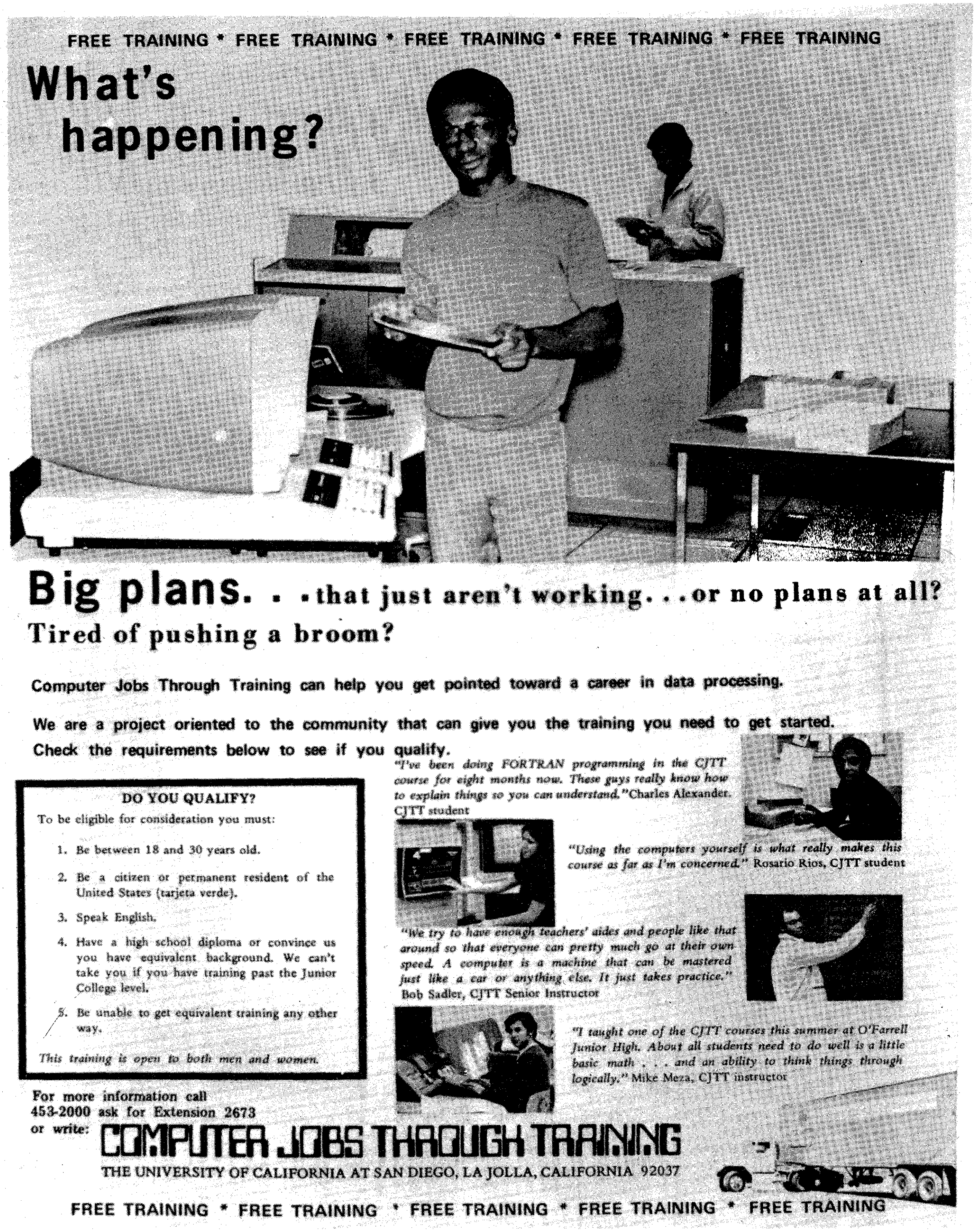
Student recruitment efforts in this environment posed several problems. Normal methods of communications such as the newspaper, radio, and television were used by our program, but proved ineffective against a widespread and unorganized disadvantaged population. A half-page ad that we ran in the local Black newspaper (Figure 6) resulted in two inquiries. Better results were obtained with an extensive poster campaign. For the most part, students learned of us "through the grapevine," and we made efforts to provide frequent input. This was particularly true in the predominantly Chicano or Mexican-American neighborhoods. In San Diego there appears to exist a time lag of about three months in the effect of this type of recruitment advertising.

Despite an admissions examination and careful in-depth interviews, we found it exceedingly difficult to assess the background and preparation of many students. Frequently, we found that students withheld information on previous educational experience out of fear that they would be deemed over qualified and be excluded from the program. The result was that in our vocational courses we tended to end up with a slightly better prepared class group than we had anticipated, representing a group which we have come to characterize as "soft core" disadvantaged. In San Diego this is really not too surprising, since the bulk of the disadvantaged population falls in this group.

Education for the disadvantaged differs from normal education in that greater sensitivity must be developed toward understanding the external forces acting on each student. It is not enough to have a well-thought-out program that excites and motivates students. We found that students who had the most promise and commitment often had to leave the course for lack of a way to support themselves. In California, it is impossible, in most cases, to be eligible for welfare support and be enrolled in a program that would lead to eventual employment. At the same time, extensive efforts on the part of the Computer Jobs Through Training staff to secure stipended support for our students proved absolutely fruitless. We found that, in fact, it was easier to initiate support for a student who wished to attend the University of California than it was to enable a person, through job training, to free himself from the dependence upon welfare agencies for subsistence. Indeed, on a couple of occasions we did precisely this.

FUNDING COMPLICATIONS

The single most overriding problem which has characterized this project almost from its beginning has



FREE TRAINING * FREE TRAINING * FREE TRAINING * FREE TRAINING * FREE TRAINING

What's happening?

Big plans. . . that just aren't working. . . or no plans at all?

Tired of pushing a broom?

Computer Jobs Through Training can help you get pointed toward a career in data processing.

We are a project oriented to the community that can give you the training you need to get started.

Check the requirements below to see if you qualify.

DO YOU QUALIFY?

To be eligible for consideration you must:

1. Be between 18 and 30 years old.
2. Be a citizen or permanent resident of the United States (tarjeta verde).
3. Speak English.
4. Have a high school diploma or convince us you have equivalent background. We can't take you if you have training past the Junior College level.
5. Be unable to get equivalent training any other way.

This training is open to both men and women.

"I've been doing FORTRAN programming in the CJTT course for eight months now. These guys really know how to explain things so you can understand." Charles Alexander, CJTT student

"Using the computers yourself is what really makes this course as far as I'm concerned." Rosario Rios, CJTT student

"We try to have enough teachers' aides and people like that around so that everyone can pretty much go at their own speed. A computer is a machine that can be mastered just like a car or anything else. It just takes practice." Bob Sadler, CJTT Senior Instructor

"I taught one of the CJTT courses this summer at O'Farrell Junior High. About all students need to do well is a little basic math . . . and an ability to think things through logically." Mike Meza, CJTT instructor

For more information call 453-2000 ask for Extension 2673 or write:

COMPUTER JOBS THROUGH TRAINING

THE UNIVERSITY OF CALIFORNIA AT SAN DIEGO, LA JOLLA, CALIFORNIA 92037

FREE TRAINING * FREE TRAINING * FREE TRAINING * FREE TRAINING * FREE TRAINING

Figure 6—Copy of recruitment material used in posters and newspaper advertising. This particular poster was used principally in the Black community. A similar poster was developed for use in predominantly Chicano communities

been the inadequacy of funding support. As indicated in Figure 2, there have been periods of many months during which meeting even the most basic operating costs required a constant juggling act and the full-time attention of most of the senior staff members. The piecemeal funding we have experienced is nothing unique to this one project. We have prepared literally dozens of proposals for funding agencies at the Federal, State, and local levels and dealt with hundreds of people in all phases of the system which ministers to the needs of the poor in this country. Piecemeal funding, interspersed with frequent, but random, periods of no funding, is the almost universal norm.

Without long-term continuity in program support, it becomes impossible to perform many functions efficiently . . . if at all. Follow-up activities for placement get interrupted, new courses cannot be started because one can't see far enough ahead to assure their completion. Funding agencies require detailed numerical performance objectives, then fund the program at an entirely different level than was requested, or many months later than anticipated. Follow-on funding required for continuity in student services is interrupted, delayed, or never materializes. Funding for stipends, if available, is almost never administered by the same agency as program funds.

This chaotic situation is of course incredibly frustrating. It is also very expensive. We have conservatively estimated that between the unproductive fund-raising activities of our senior staff, the periods of "tooling up" for major new efforts, and the periods we have operated at reduced capacity because we could not afford a larger effort, a full third or more of our total expenditures might have been saved. That is to say, that our total productivity has been only something like 60 percent of what it might have been had we had the same resources available to us in a long-term continuous manner to expend when and where they were needed over a two- or three-year period.

We have consequently been forced to the reluctant conclusion that this kind of funding reflects a fundamental structural weakness in the manner in which poverty program activities are funded in this country . . . one which we do not expect will be corrected within the next few years. Unless they enjoy some very special financial arrangement, we suspect that future programs such as ours are doomed to an almost equivalent funding experience.

CONCLUSIONS

Three years of experience in developing and running the Computer Jobs Through Training program have led us to the following conclusions:

- Computer instruction for the disadvantaged is a powerful tool for reaching and "turning on" students who have previously shown only limited academic interests and capabilities. It appears to be most successful as a motivational tool with younger age groups. At the level of vocational training for young adults, success appears to be largely restricted to the "soft core" disadvantaged but this restriction is essentially absent in pre-vocational and motivational courses designed for younger age groups.
- The technical details of programs in this field, that is, questions involving specific computer hardware to be used, programming language used, etc., which we discussed at great length in our earlier paper, turn out to be relatively unimportant. Interactive capability, especially during program execution, along with the ability to bring computer access directly into the students' own community or school, does appear to be a significant factor in the design of a successful program.
- Success with programs of this nature requires an unusually high level of staff skill, not just in technical subjects but also in the fields of social and community relations. Close community ties, at the neighborhood level, among spokesmen for the poverty community, in public agencies, and with local business are absolutely essential.
- Comprehensive student services are very important. At the motivational and pre-vocational levels, these principally include orientation, familiarization, and guidance in further education. At the vocational level they must include a full range of counseling services with ties to medical and legal counseling. Success at this level among truly hard-core adult populations is essentially impossible without some form of income maintenance. Funds for such income support in this kind of program are generally not available under current local, State, or Federal programs.
- Despite the widespread conviction that serious and effective training programs offer one of the most powerful tools for breaking the cycle of poverty, public and private funding for such programs is remarkably scarce. When public funding does exist, it is frequently used unwisely or in very conventional programs. Public funding for the continued operation of innovative programs has, in our experience, proved almost non-existent.

- Somewhat to our surprise, our greatest impact, despite good direct training results, has fallen in the more intangible area of horizon broadening and attitude change. Unlike most poverty programs, our project staff has been characterized by a high level of professionalism. The effect of an extended exposure to the professional environment together with the increased self confidence and self value which the project imparted to most students, has had a substantial positive impact on their outlook and motivation.

In the spring of 1972 the Computer Jobs Through Training program ceased to exist at the University of California at San Diego. Under support from the California Office of Vocational Education, arrangements were undertaken to transfer the project from the University to some other local agency more directly responsible for the student population involved. After several false starts with other agencies, a transfer was finally arranged to the San Diego County Department of Education where the program will operate in modified form under their Regional Occupational Program.

ACKNOWLEDGMENTS

We acknowledge with pride the support of Bradley Rogers, Curtis Bagby, Michel Meza, Lucy Enriquez, Susan Halfon, Josie Foulks, David MacQuoid, Oneeta Alexander, Hazel Welborne, Mary Mirabito and Victor Carranza, of the Computer Jobs Through Training Staff.

Financial support has come from the Rosenberg Foundation, from the California Office of Vocational Education, from the University of California, from Gulf Energy and Environmental Systems, and from other business and private supporters.

REFERENCES

1 M G MORGAN M R MIRABITO N J DOWN
Computer jobs through training—A preliminary project report
AFIPS Proceedings of the Fall Joint Computer Conference Vol 37 p 345 1970

2 CALIFORNIA STATE DEPARTMENT OF HUMAN RESOURCES DEVELOPMENT
Summary of employment and unemployment—San Diego metropolitan area part A 1966-1971
California HRD Research and Statistics Section PO Box 923 San Francisco California 94101

Implementation of the systems approach to central EDP training in the Canadian government

by G. H. PARRETT and A. K. PRAKASH

Government of Canada
Ottawa, Ontario, Canada

INTRODUCTION

This paper describes the approach taken in introducing a central training program into the federal government of Canada. The task represented a challenge faced by all large, complex and multi-faceted organizations and the experiences of the Canadian government should be of interest to all who have a concern for training in electronic data processing.

To illustrate the nature of the organization being treated we remind the reader that Canada is second in geographical size among the nations of the world, exceeded only by the Republic of China. It extends from the Atlantic to the Pacific coasts and from the 49th parallel to the Arctic circle with territory totalling 3,852,000 square miles. The population at latest census was just at the 22 million mark with the bulk of the residential centers found in a narrow ribbon just north of the U.S.A.-Canada border.

To provide the services needed by this population the Canadian government is organized into several departments and agencies, each of which is autonomous although they are linked through the medium of the Public Service Commission, which coördinates most personnel functions, and a Treasury Board Secretariat which acts as general manager for the Public Service and sets the policy for its administration. There are approximately 55 independent departments and agencies and together they employ approximately 203,000 public servants.

The annual expenditure on EDP in the Canadian government is estimated to be about $50,000,000 with an annual increase expected to range from 15 to 20 percent. Because of the wide range of services demanded by people in a democratic nation, there is, throughout the Canadian government, a multiplicity of computer configurations, each chosen by the pertinent department or agency as the best medium for assisting it to fulfil its role. To provide the personnel to utilize this computer investment there are 1200 Computer Systems Administrators (systems analysts and/or programmers) and 1800 Data Administrators (operators). In Figure 1 we have shown 6 sample departments showing their different needs and the type of configuration each has installed to meet the demands placed upon it.

We see that the Canadian government consists of a number of sub-organizations, each possessing considerable autonomy of action and most having a computer need different from any other. In such a situation the inevitable jungle of disjointed and uncoordinated training activities emerged. Those departments with major computer installations instituted EDP training schools of their own and tailored their activities to meet the particular needs of their own department. For training of management personnel with little technical knowledge, departments relied upon courses advertised in the commercial sector. Much of the training, both for non-technical personnel and for computer specialists, was manufacturer oriented and some needed training was left uncovered because ready-made courses were not available from commercial suppliers. On the other hand, there was much duplication of effort as departments often expended resources on a training activity already developed and available in another. Despite these problem areas a great deal of useful training took place but the positive results were weakened by the fact that little, if any, standardization existed from department to department.

The increasing use of computers year by year and the accompanying demands for ever greater training expenditures pointed out the need for coordination and standardization of the total training activity. To this end, the Treasury Board Secretariat instituted a study to develop a comprehensive data processing training

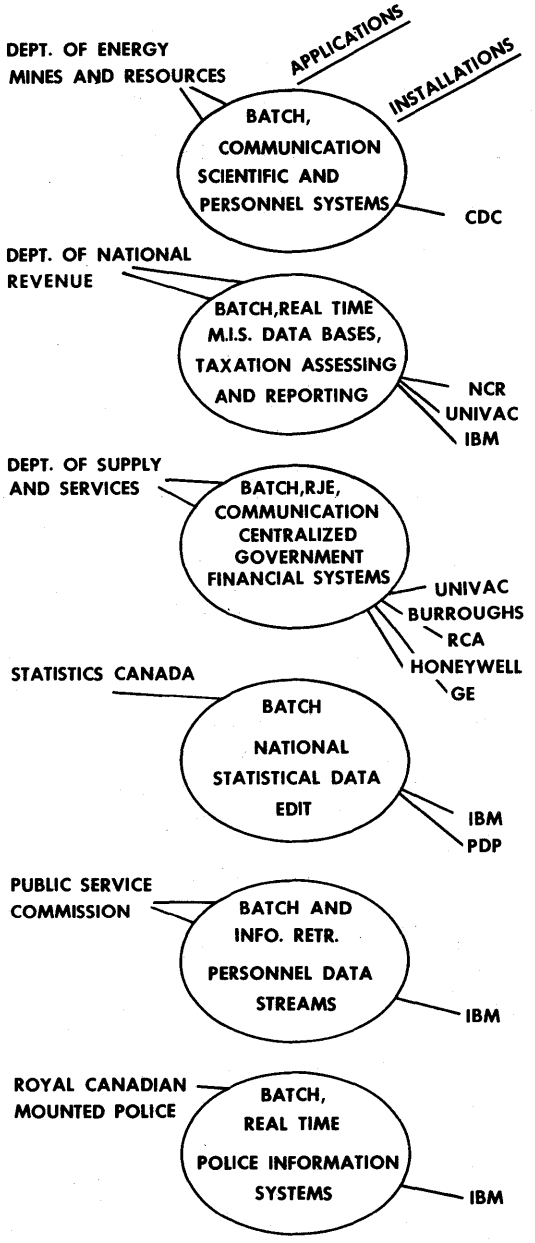


Figure 1—Representative departments

program for the federal government. The steps involved in this study and the approach taken in the implementation of such a program are the subject of this paper.

FEASIBILITY STUDY

The Treasury Board initiated its study in 1968 and set the following as the terms of reference for the desired central training program.

(a) To raise the general level of competence in all areas of EDP activity within the government.
(b) To bridge the communication gap between users and EDP specialists.
(c) To alleviate the recruiting retention and internal mobility problems in the EDP area.

In organizing their examination and the program development, four phases were established:

PHASE 1—PRELIMINARY DESIGN

The main purpose of this phase was to identify data processing training needs within the government and to design the required training program in sufficient detail that cost and resource estimates for program implementation and operation could be formulated.

PHASE 2—COST AND IMPLEMENTATION

During this phase estimates of the numbers of personnel to be trained were developed and the cost, personnel and other resources required to carry out detailed program design and program operation and maintenance were determined.

PHASE 3—DETAILED PROGRAM DESIGN

When authority was granted to proceed with the program, on the basis of Phases 1 and 2, detailed program design was carried out. Action was taken to prepare, or obtain, detailed course contents, teaching aids, student manuals, instructors' guides and other course materials for each course in the training program.

PHASE 4—PROGRAM OPERATION AND MAINTENANCE

The purpose of this phase was to carry out the training and to review and revise the training program as requirements change.

The Treasury Board Secretariat assumed direct responsibility for Phase 1 but were careful to ensure that the initial examination reflected the views of the many diverse interests in EDP training. Accordingly, the study team included management consultants experienced in computer sciences, trainers, representatives of

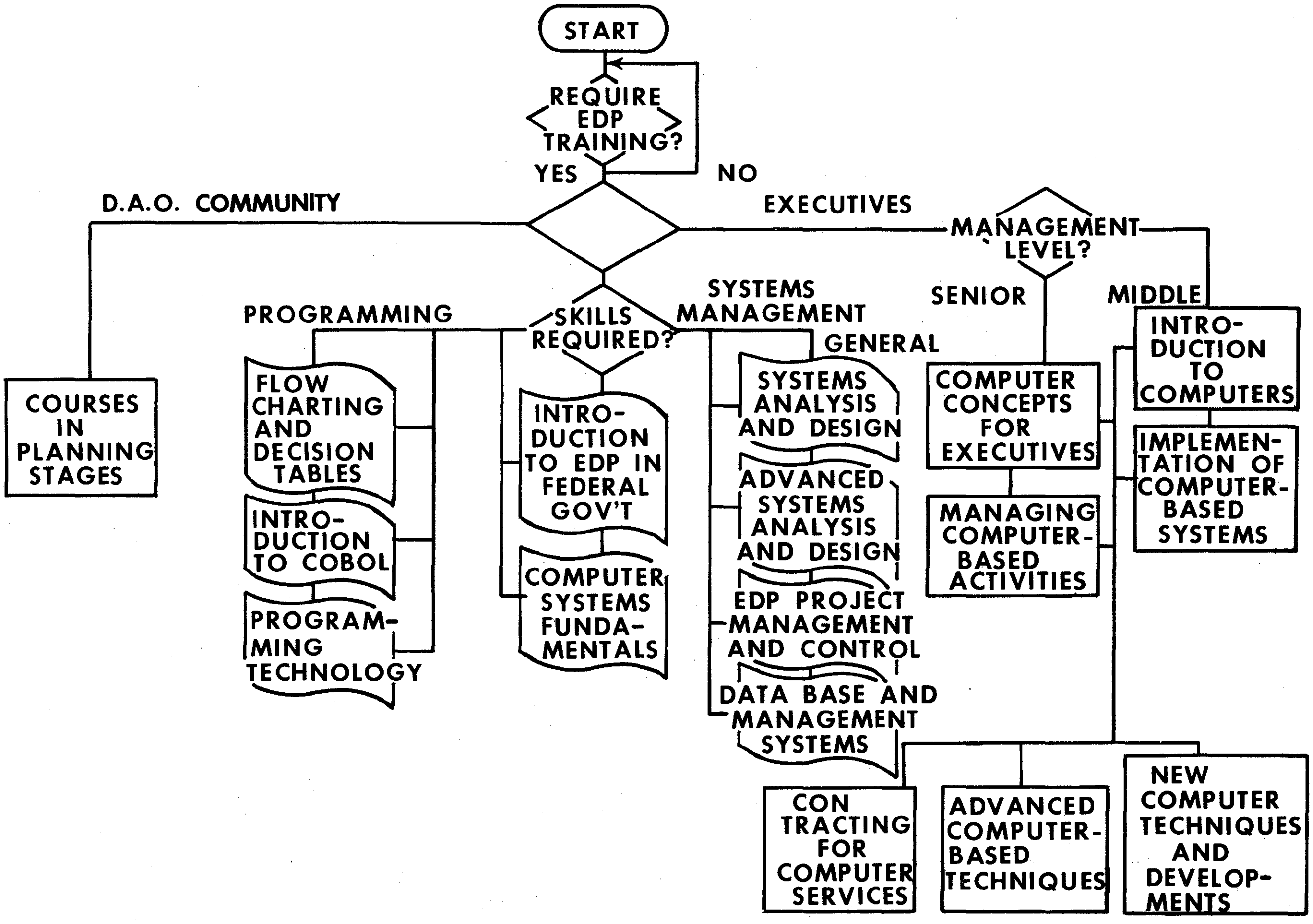

START
REQUIRE EDP TRAINING?
YES
NO
D.A.O. COMMUNITY
EXECUTIVES
MANAGEMENT LEVEL?
SENIOR
MIDDLE
PROGRAMMING
SKILLS REQUIRED?
SYSTEMS MANAGEMENT
GENERAL
COURSES IN PLANNING STAGES
FLOW CHARTING AND DECISION TABLES
INTRODUCTION TO COBOL
PROGRAMMING TECHNOLOGY
INTRODUCTION TO EDP IN FEDERAL GOV'T
COMPUTER SYSTEMS FUNDAMENTALS
SYSTEMS ANALYSIS AND DESIGN
ADVANCED SYSTEMS ANALYSIS AND DESIGN
EDP PROJECT MANAGEMENT AND CONTROL
DATA BASE AND MANAGEMENT SYSTEMS
COMPUTER CONCEPTS FOR EXECUTIVES
MANAGING COMPUTER-BASED ACTIVITIES
INTRODUCTION TO COMPUTERS
IMPLEMENTATION OF COMPUTER-BASED SYSTEMS
CON TRACTING FOR COMPUTER SERVICES
ADVANCED COMPUTER-BASED TECHNIQUES
NEW COMPUTER TECHNIQUES AND DEVELOPMENTS

Figure 2—Central EDP program courses

user departments and personnel concerned with the staffing implications. A report was submitted in March, 1969 and following its acceptance the Treasury Board authorized the Public Service Commission to proceed with Phase 2 on cost and implementation.

The Preliminary Design recommended a training program with three series of courses, each series designed to meet the particular needs of a designated group of personnel. The divisions recommended were a Management Series for non-EDP management personnel, a Systems Series for Computer Systems Administrators and an Operators Series for Data Administrators. A flow chart showing the proposed outline training plan is shown in Figure 2.

CENTRAL EDP TRAINING PROGRAM

When the Public Service Commission was authorized to proceed with further development it designated the Bureau of Staff Development and Training as its instrument and the Central EDP Training Program was formed.

The program required personnel who combined computer science expertise with training experience. Such a staffing commitment proved to be a difficult assignment but over a period of time the needed personnel were recruited, some from within the public service while others came from educational institutions and commercial houses. To provide an avenue for personal development for these staff members and to ensure that the program remained aware of operational developments, a rotational staffing system was planned. The recruiting arrangement was for each staff member to join the program for a minimum of one year and a maximum of two years, at which time they would return to the operational element of the computer community within the government and new staff found for the program.

SYSTEMS APPROACH

The Bureau of Staff Development and Training adopts a "Systems" approach in developing and conducting the indicated training. In this way the desired terminal behavior is identified in the first instance and the training needed to effect this behavior is then introduced. Briefly, the "Systems" approach follows ten steps:

1. Analyze skills required to carry out job requirements at each job level for computer systems personnel.
2. Based on the above analysis, specify learning objectives in terms of skills required that a course should achieve.
3. Solicit input and criticism from the computer personnel of government departments regarding the proposed objectives.
4. Modify course objectives in harmony with the requirements of government departments and develop detailed course standards.
5. Conduct feasibility studies regarding the development of the course meeting established standards. If resources within or external to the government can economically offer a course meeting Bureau Standards, contracts are let on an individual course basis.
6. Develop student manuals, instructors' guides and other materials.
7. Exercise control over any course to ensure that development meets Bureau standards.
8. Conduct the course with material developed under the sponsorship of the Bureau.
9. Evaluate the effectiveness of training in the course.
10. Modify the course where required in order to meet objectives and standards.

Many of the advantages of this approach to training are self evident. In particular, this method allows the training agency to establish quantified learning objectives for each course element. Moreover, it draws the user of the end product into active participation in course development through his input and criticism of the individual learning objectives and the specifications and standards for each course. Finally, it allows for continuous improvement and up-dating of courses, to meet the NEEDS of the users.

A mechanism to ensure that the central program remains sensitive to the needs of the community it serves is provided by an Interdepartmental EDP Advisory Committee. The members of this group come from several of the user government departments and meet regularly to advise the central program on the ways in which the computer community could best be served. This committee reviews all proposed course specifications and standards and in this way the training objectives are kept tuned to government operations.

TABLE I—Management Series

M1-INTRODUCTION TO COMPUTERS	3 DAYS
M2-COMPUTER CONCEPTS FOR EXECUTIVES	2 DAYS
M3-MANAGING COMPUTER-BASED ACTIVITIES	3 DAYS
M4-IMPLEMENTATION OF COMPUTER-BASED ACTIVITIES	3 DAYS
M5-CONTRACTING FOR COMPUTER SERVICES	1 DAY
M6-ADVANCED COMPUTER-BASED TECHNIQUES	2 DAYS
M7-NEW COMPUTER TECHNIQUES AND DEVELOPMENTS	1 DAY

For any training to be viable a continuing evaluation must be made of the effectiveness of the instruction it provides. The Bureau uses three techniques to help it

TABLE II—Systems Series

S2-COMPUTER SYSTEMS FUNDAMENTALS	5 DAYS
S3-FLOWCHARTING AND DECISION TABLES	5 DAYS
S4-INTRODUCTION TO COBOL	10 DAYS
S5-PROGRAMMING TECHNOLOGY	10 DAYS
S6-ELEMENTS OF SYSTEMS ANALYSIS AND DESIGN	10 DAYS
S7-EDP PROJECT MANAGEMENT AND CONTROL	5 DAYS
S8-ADVANCED SYSTEMS CONCEPTS	5 DAYS
S9-DATA BASE AND MANAGEMENT SYSTEMS	3 DAYS

assess the value of its efforts and to contribute to modification and up-dating of its courses:

(A) COURSE CRITIQUE
At the end of any course each participant is asked to complete a course critique. This will reflect whether, in the participant's opinion, the course met its objectives.

(B) COURSE MONITOR
One or more members of the Interdepartmental EDP Advisory Committee are asked to attend the presentation of each course, particularly during its initial serial. His report will indicate, through his experienced eyes, whether the course provided the needed training.

(C) EVALUATION SERVICE
The Bureau of Staff Development and Training has a formal evaluation service that uses trained psychologists several months after course completion to evaluate the effectiveness of the training. Their objective is to determine if there has been an improvement in the use of skills that is attributable to the course.

COST RECOVERY

The Bureau operates its programs on a cost recovery basis. Under this system the cost of developing and conducting the courses are carefully calculated and tuition fees established to return offsetting revenue. By charging for the service and requiring actual payments to be made a more efficient and effective accountability for and utilization of this common service is achieved and greater cost consciousness and identification of total costs results. The customer departments are made more aware of the value of the training and are careful to accurately determine their needs. On the other hand, the program management is directed toward maximum efficiency since they must remain competitive with similar, although non-government oriented courses in the private sector.

COURSE STRUCTURE

The Management series of courses was designed to encourage the serious involvement of non EDP management in the problems, decisions and actions related to computer projects. In Table I is shown the array of courses offered.

This series of courses form a continuum of EDP training for management, all treating with the computer in a government environment. Although each course may be enjoyed independently, together they form a progressive training program. The continuum is shown in Figure 3, in graph form.

TABLE III—Operators Series

COMPUTER FUNDAMENTALS	**3 DAYS**
DATA PROCESSING TECHNOLOGY	**3 DAYS**
INTRODUCTION TO PROGRAMMING SYSTEMS	**5 DAYS**
COMPUTER SYSTEMS SUPERVISION	**3 DAYS**
INTRODUCTION TO COMPUTER SYSTEMS ADMINISTRATION	**2 DAYS**

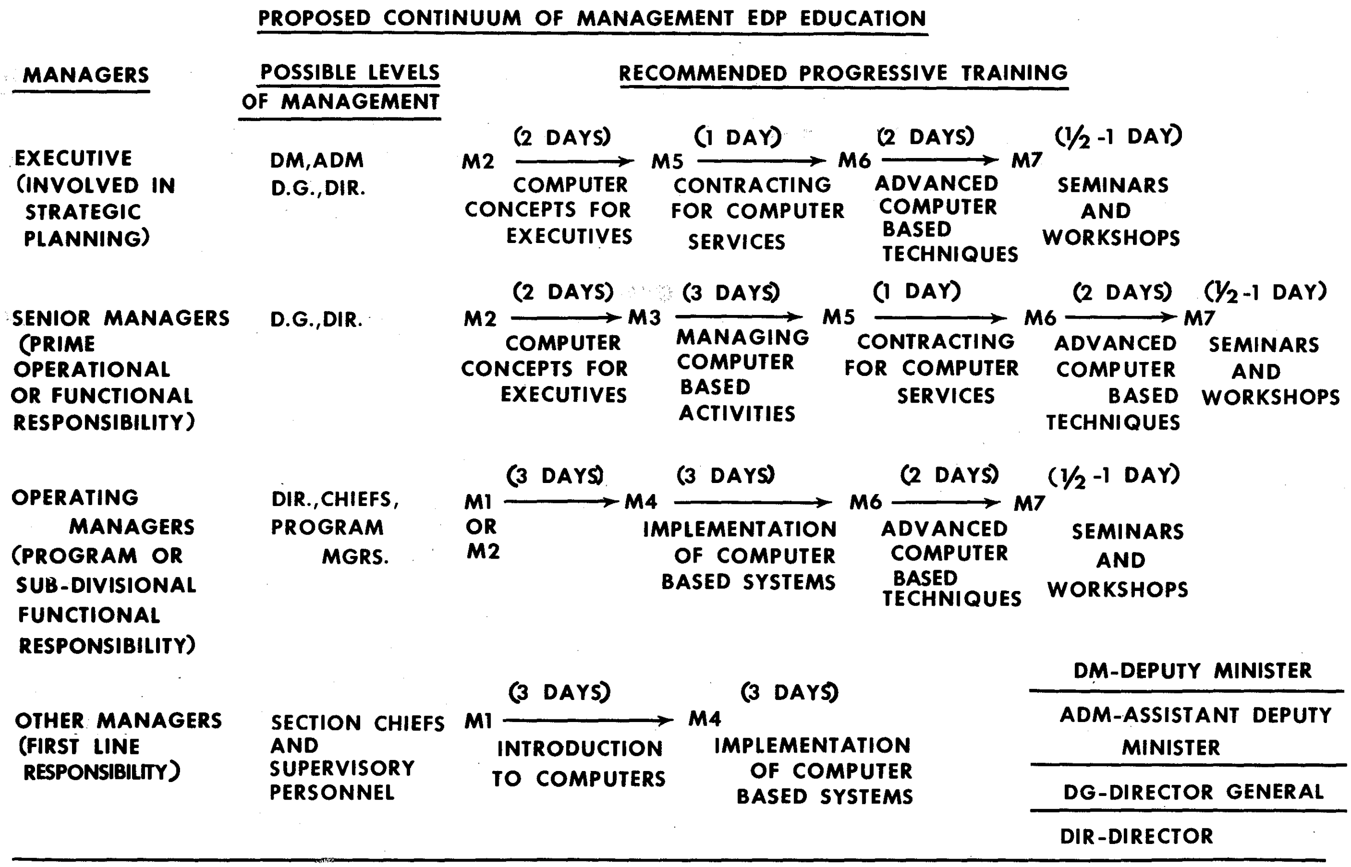
The Systems Series was developed to provide a mechanism for expanding and up-dating the technical skills of systems analysts and programmers in the various elements of computer sciences. Table II lists the courses making up this series.

The Operators Series is seen to be a two phase requirement. In the initial phase the theory, principles and concepts of data processing technology, its supervision and administration will be covered. This series of courses will be non-machine oriented with its aim to allow the operator to identify his position in the total computer activity and to improve his ability to supervise and administer computer operations. The courses are being developed this training year for introduction in 1973/74, and they are listed in Table III.

The second phase of the Operators Series instruction will be related to individual computer configurations and problems of implementation. This phase will be developed after the courses in the initial phase have been introduced and a thorough study of actual needs has been carried out.

CONCLUSION

The Central EDP Training Program of the Canadian government is seen to be a sound program which offers STANDARDIZED training to meet the NEEDS of the USERS. It covers a wide range of interests for Managers, Systems Analysts and Programmers and Computer Operators. The essential value of the program lies in its system of INVOLVING the end user of the training in the design, development, evaluation and modification of the courses offered.



PROPOSED CONTINUUM OF MANAGEMENT EDP EDUCATION

MANAGERS	POSSIBLE LEVELS OF MANAGEMENT	RECOMMENDED PROGRESSIVE TRAINING
EXECUTIVE (INVOLVED IN STRATEGIC PLANNING)	DM,ADM D.G.,DIR.	M2 (2 DAYS) COMPUTER CONCEPTS FOR EXECUTIVES → M5 (1 DAY) CONTRACTING FOR COMPUTER SERVICES → M6 (2 DAYS) ADVANCED COMPUTER BASED TECHNIQUES → M7 (½-1 DAY) SEMINARS AND WORKSHOPS
SENIOR MANAGERS (PRIME OPERATIONAL OR FUNCTIONAL RESPONSIBILITY)	D.G.,DIR.	M2 (2 DAYS) COMPUTER CONCEPTS FOR EXECUTIVES → M3 (3 DAYS) MANAGING COMPUTER BASED ACTIVITIES → M5 (1 DAY) CONTRACTING FOR COMPUTER SERVICES → M6 (2 DAYS) ADVANCED COMPUTER BASED TECHNIQUES → M7 (½-1 DAY) SEMINARS AND WORKSHOPS
OPERATING MANAGERS (PROGRAM OR SUB-DIVISIONAL FUNCTIONAL RESPONSIBILITY)	DIR.,CHIEFS, PROGRAM MGRS.	M1 OR M2 (3 DAYS) → M4 (3 DAYS) IMPLEMENTATION OF COMPUTER BASED SYSTEMS → M6 (2 DAYS) ADVANCED COMPUTER BASED TECHNIQUES → M7 (½-1 DAY) SEMINARS AND WORKSHOPS
OTHER MANAGERS (FIRST LINE RESPONSIBILITY)	SECTION CHIEFS AND SUPERVISORY PERSONNEL	M1 (3 DAYS) INTRODUCTION TO COMPUTERS → M4 (3 DAYS) IMPLEMENTATION OF COMPUTER BASED SYSTEMS

DM-DEPUTY MINISTER

ADM-ASSISTANT DEPUTY MINISTER

DG-DIRECTOR GENERAL

DIR-DIRECTOR

Figure 3—Canadian Government Central EDP Training Program

Evaluations of simulation effects in management training

by H. A. GRACE

University of Southern California
Los Angeles, California

INTRODUCTION

The management laboratory, in which the training and evaluation reported in this paper takes place, is designed to accommodate persons in sets of between five and ten. Persons primarily using the laboratory are undergraduate and graduate students, and prospective or practicing managers.

The laboratory encourages utilization of many techniques, including both manual and computer-based simulations, as means for improving trainees' task accomplishment and team development. Other techniques include lectures, films, case analyses, and discussions. Simulations include manual exercises designed specifically for the management laboratory, computer-based games, improvisations employing videotape for immediate feedback, and both trainee-developed and commercial exercises and games.

METHOD AND RESULTS

During the conduct of a commercial game being played by two sets of trainees, we observed categorically opposite effects on each set. To test our hunch, we hastily designed, dittoed, and administered an assessment form which asked the trainees how well their teams had developed prior to the simulation and after it; why they had selected that game; and to what they attributed the game's effects. Table I reports the effects as perceived by these two sets of trainees.

As we had observed, trainees actually experienced almost polar differences as a result of playing this commercial game, and the effects were primarily a function of their team's pre-simulation cohesiveness. Some of the comments made by trainees in the developed team were: "It produced uneasy feelings; I felt uncertain and vindictive; nobody followed the rules; there was deceit and confusion; I felt let down afterwards; because of the double-crossing, I couldn't even trust my partner." Trainees in the underdeveloped team wrote: "It opened my mind more to others and made me forget my inhibitions; I really got involved for the first time; the decision-making and bargaining helped us; it brought out a desire to compete and win; people can really manipulate when it's beneficial." Interestingly enough, members of the more developed team were so disturbed by the game's effects that they immediately replayed it, assuming different roles than before, and attempted to regain their prior degree of cohesion. Further investigations of this kind revealed a variety of reasons for selecting a simulation: "It was a challenge; we heard it was fun; we wanted something that would involve everyone; we thought that working with different partners would help; it looked like it would help us make advances on our task." Some of these reasons expressed expectations about the task, some about oneself, and some about the team.

After a series of iterations, the Event Assessment, shown as Table II, was generated for use in evaluating the effects of any simulation by persons engaged in management training. Our justification for using this generalized approach is that we experienced disparate effects from the collecting of such assessments, as well as from feeding back the results of assessments to trainees. Because training events occur so fast, often without their being planned, the Event Assessment uses past, present, and future tenses in many questions.

Scoring, tabulating, analyzing, and presenting the results of an event assessment can be accomplished manually or by a computer program. We have not as yet explored the effects of immediate versus delayed feedback of results, nor have we always postponed trainees' choice of a simulation until its anticipated effects have been fully explored. Data collection has been both facilitated and standardized by means of this form.

Results from the Event Assessment Form may be displayed in various ways in order to facilitate manage-

TABLE I—Perceived Effects of a Simulation on Team Development

TEAM DEVELOPMENT		POST-SIMULATION	
PRE-SIMULATION	LITTLE	SOME	GREAT
GREAT	DDDD	D	UU
SOME		U	DUU
LITTLE		UUU	UU

D: trainee in developed team
U: trainee in undeveloped team

ment training objectives. Table III, for instance, indicates how one set of trainees perceived a simulation's effects on the accomplishment of the task assigned to it. Table IV displays how this same set of trainees experienced that simulation's effects on the development of the team itself. Table V combines task and team information, and serves also as an indirect sociometric device. The power of these combinations of data lies in the trainer's ability to focus trainees' attention on the various tradeoffs between task accomplishment and team development which a given simulation or other event stimulates.

TABLE II—Event Assessment Form

Assessor:________________Date:_________Time:__________
Name of Event:_________________________________
PRE-EVENT QUESTIONS
What is/was your task? To_______________________
How well are/were you accomplishing your task?

VERY POORLY	POORLY	SO-SO	WELL	VERY WELL

How well are/were you developing as a team?

VERY WELL	WELL	SO-SO	POORLY	VERY POORLY

ABOUT-THE-EVENT: How much do/did you expect the event to: Help you make progress on your task?

VERY MUCH	MUCH	SOME	LITTLE	VERY LITTLE

Help you all develop as a team?

VERY LITTLE	LITTLE	SOME	MUCH	VERY MUCH

POST-EVENT: In what way and how much will/did the event: Affect your progress on your task?

VERY POSI-TIVELY	POSI-TIVELY	SO-SO	NEGA-TIVELY	VERY NEGA-TIVELY

Why? Because_________________________________
Affect your development as a team?

VERY NEGA-TIVELY	NEGA-TIVELY	SO-SO	POSI-TIVELY	VERY POSI-TIVELY

Why? Because_________________________________

TABLE III—Perceived Effects on Task Accomplishment

TASK ACCOMPLISHMENT			POST-SIMULATION		
PRE-SIMULATION	VERY LITTLE	LITTLE	SOME	GREAT	VERY GREAT
VERY GREAT		B			
GREAT	A	CE		K	L
SOME					
LITTLE				J	M
VERY LITTLE		DF		GH	

As Table III shows, four of the six people who felt great or very great pre-simulation task accomplishment, also felt that the simulation had little or very little ef-

TABLE IV—Perceived Effects on Team Development

TEAM DEVELOPMENT			POST-SIMULATION		
PRE-SIMULATION	VERY LITTLE	LITTLE	SOME	GREAT	VERY GREAT
VERY GREAT	A				
GREAT		BC		HJK	LM
SOME					
LITTLE	D	E			
VERY LITTLE		F			G

TABLE V—Perceived Effects on Task Accomplishment and Team Development

TASK ACCOMPLISHMENT	TEAM DEVELOPMENT		
	LOSS (−4 to −2)	SAME (−1 to +1)	GAIN (+2 to +4)
GAIN (+2 to +4)		HJM	G
SAME (−1 to +1)		DFKL	
LOSS (−2 to −4)	ABC	E	

fect, (ABCE). On the other hand, four of the six who experienced little or very little pre-simulation task accomplishment felt that the simulation had great or very great effect (GHJM).

Table IV indicates that five of the eight persons who thought their pre-simulation team development was great or very great also considered the simulation's effect to be great or very great (HJKLM). Only one of the four persons who experienced little or very little pre-simulation team development, however, perceived the simulation as having a great or very great effect (G).

The data in Table V show how trainees experienced effects of gain and loss in both task accomplishment and team development as a result of the simulation. For the most part, trainees associate task and team experiences, as indicated by the linear relationship in Table V. These data, however, also permit a sociometric analysis of the participants. By their similar perceptions, trainees A, B, and C constitute a cluster which experienced a double loss in both task accomplishment and team development, while member G stands alone, having experienced a double gain. Discussion about these data centered on whether both effects might not be an example of regressing toward the mean. Four members experienced no change as the result of the simulation, (DFKL), while members H, J, and M preceived a gain in task accomplishment only, and one member, E, a loss in task accomplishment.

The display of this information, fed back to management trainees, gives everyone a better picture of the array of effects of any event, including a simulation, on both the task and the team. Regardless of the simulation's other objectives, these results can be useful for purposes of management development.

Conceptual design of an eight megabyte high performance charge-coupled storage device

by B. AUGUSTA and T. V. HARROUN

IBM Components Division
Essex Junction, Vermont

SUMMARY

The design approach suggested to satisfy the conceptual requirements was the use of self-contained, charge-coupled storage chips with on-chip decoding. In this approach, the information on the memory chip is stored in a group of closed-loop shift registers, and random access is provided to any one of the registers by an on-chip dynamic FET decoder. In this way, n-control lines can select one of 2^n shift registers.

Of the many organizations possible in expanding the on-chip decoding concept into a design for a 10^8-bit memory, a bit-per-chip organization was chosen. This proposed organization results in reasonable chip power dissipation and was contained in successively higher levels of packaging. The operating characteristics are summarized in Table I.

The models, limited in capacity to that necessary to show feasibility of the approach used, are intended to demonstrate the operability of the conceptual design. The operations necessary to perform the functions of the storage chip are charge injection, charge transfer efficiency, sensing, absence of channel cross-talk, turn around and charge generation.

These are all performed using a silicon self-aligned gate structure driven by a four-phase electric field, and are basic to the operation of the shift registers. Decoders which select one of the 2^n shift registers have been amply explored by industry.

Two small shift register structures were designed, fabricated and tested to demonstrate these functions and concepts. One, a 480-bit register demonstrating injection, sensing, amplification and turnaround, was designed using 0.3-mil line widths. The second, a 256-bit shift register demonstrating high density, charge retention, and absence of cross talk, was designed using 0.15-mil line widths. Operation is summarized in Table II.

The conceptual design and the feasibility models are described in considerably more detail in the following sections.

GENERAL CCD MASS STORAGE DESIGN

This section describes the conceptual design of a 10^8-bit charge-coupled device mass storage unit. The design principles of the chip, module, system, and error correction and detection are discussed first.

The system is designed to be used in a block-oriented mode, in which random access is provided to blocks of information, which are then read out or written in serially. Physically, this means that the CCD chip is divided into a number of closed-loop shift registers that store the blocks of information. Random access to these shift registers is provided by on-chip FET decoders. Propagation in the shift registers, which occupy the major portion of the chip, is accomplished with a two-level interconnection pattern activated by an external four-phase electric field.

The further organization of such chips into a memory system depends on a number of design criteria. Since the application contemplated here was industrial, low cost and operational life were emphasized. In addition, the storage system was required to be of small size and low power and to operate in typical industrial environments without elaborate support. Consequently, the chip design should give reasonable chip yields, the package should be reasonably easy to manufacture, and the means of achieving the reliability should not be prohibitively expensive fraction of the total storage. These topics are considered in the next several sections. The one remaining task of this section is to specify the access time and data rate of the storage system. After examining present and future needs, it was decided to aim for average access times below 0.5×10^{-3} sec and

TABLE I—Characteristics of the Design of a 10^8-Bit Memory

Capacity	1.0×10^6 words of 73 bits each
Shift frequency	500 kHz
Data rate	32×10^6 bits/sec
Access time	256 μsec to a block of 256 words 512 μsec to a word
Power	237 watts

data transfer rates of at least 3×10^6 bytes/sec. The next few sections describe the influence of these requirements on the design evolution.

Chip, module and card design

Chip size

It is desirable to have as many bits per chip as possible, consistent with reasonable chip yield. In addition, high speed and low cost are easier to obtain as the density of information storage (bits/square cm) increases. These are the primary considerations in choosing the size of the chip.

Based on fabrication experience and on progress in the industry, it seems reasonable to assume that layout ground rules with 0.15-mil minimum line widths will be practical for the mid-seventies. Assuming that a four-phase structure is used to store a bit of information and that information flow is reversed between adjacent parallel channels to close a shift register, topological considerations yield a storage cell area of 15.4 w^2, where w is the minimum line spacing. This corresponds to a storage density of $\sim 2.9 \times 10^6$ bits per square inch. With such densities, a 32,768 bit chip might to be feasible.

Chip layout

The chip organizational layout is uniquely determined by the average access time that the system is to have. Except for some relatively small decoder delays, the average access time T_A is given by

$$T_A = \tfrac{1}{2} B_{SR} T_S$$

where B_{SR} is the number of bits per shift register and T_S is the time period of one shift. The shift frequency is chosen to be 500 kHz to minimize chip power and allow variable speed operation. Hence, $T_S = 2 \times 10^{-6}$ sec. It was stated earlier that T_A should be less than 0.5×10^{-3} sec. Therefore, B_{SR} must be less than 500 bits. The nearest binary number is 256 bits so the proposed chip design calls for 128 shift registers of 256-bits. This results in an average access time of 256 μsec plus a small amount of decoder and other delays.

TABLE II—Measured Operating Characteristics of Feasibility Models

Material	Silicon, n-type substrate
Structure	Modified self-aligned gate, 2-level interconnection
Cell size	0.32 mil^2
Phase voltages	10-12 volts
Output signal	4-volt amplitude
Charge retention	> 0.25 sec
Minimum shift rate	5 kHz

The chip layout is shown schematically in Figure 1. Each of the 128 closed-loop shift registers has an input-output amplifier, four restore amplifiers and a select decoder. These are organized into four nearly square groups of 32 shift registers, allowing shortened phase lines to drive from the center of the chip. Each shift register is folded four times to match I/O amplifier pitch. A restore amplifier follows each 64-bit register segment.

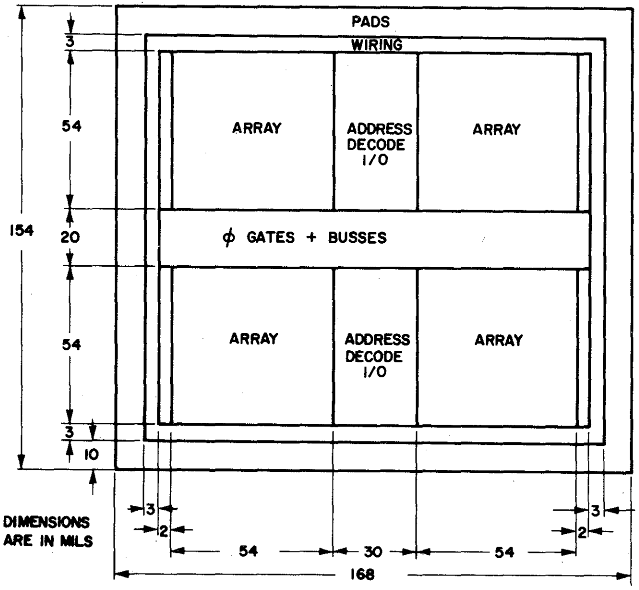


Figure 1—Proposed memory chip layout

The relative areas occupied by each function on the chip are defined in the following way. A shift register is the product of the vertical and horizontal periods of the cell. The storage section of each array contains 32 shift registers, occupying an area of

$$(64 \times 5.3 \text{ w}) \times (32 \times 4 \times 2.89 \text{ w}) = 12{,}600 \text{ w}^2$$
$$= 8192 \text{ cells.}$$

By comparison, each restore amplifier occupies

$16\ \mathrm{w} \times 5.8\ \mathrm{w} = 94\ \mathrm{w}^2$
$= 6$ cells and the decoder I/O amplifier occupies
$(84\ \mathrm{w} \times 11.6\ \mathrm{w}) + 2(16\ \mathrm{w} \times 2.9\ \mathrm{w}) = 1064$
$= 69$ cells.

Phase gates and bussing occupy a total area on the chip equivalent to 8200 cells. The resulting chip, including wiring and pad space, is 168 mils by 154 mils.

Register selection, writing, reading and clearing are all accomplished with 14 control lines. Phase drive, reference voltages and supply voltage require 11 lines, resulting in 25 I/O connections per chip.

Module design

The proposed module is a design extension of IBM's widely used logic and monolithic memory module. The proposed module, however, is 35 percent larger than the standard module (0.580 in. square) and provides an increased number of I/O pins. Since only one chip is active at a time, the module thermal characteristics permit two chips to be accommodated per module. Consequently, the module has two stacked ceramic substrates, with one chip per substrate (Figure 2), joined by the IBM controlled collapse solder connection.

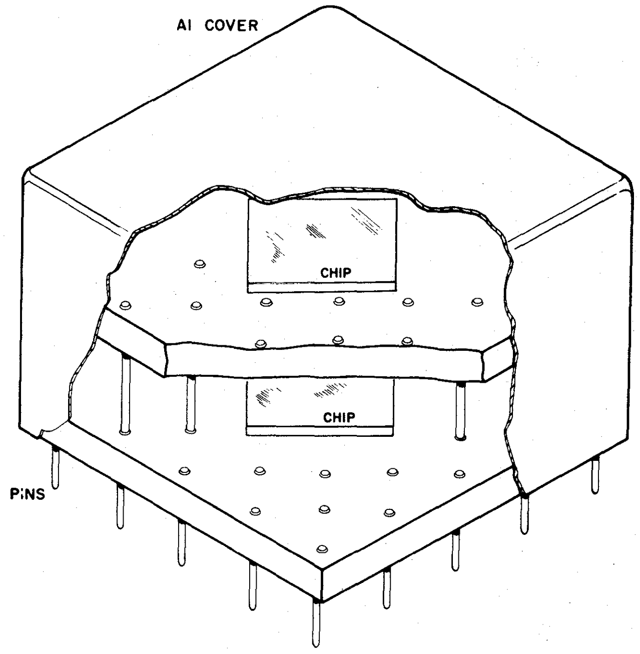


Figure 2—Basic memory module

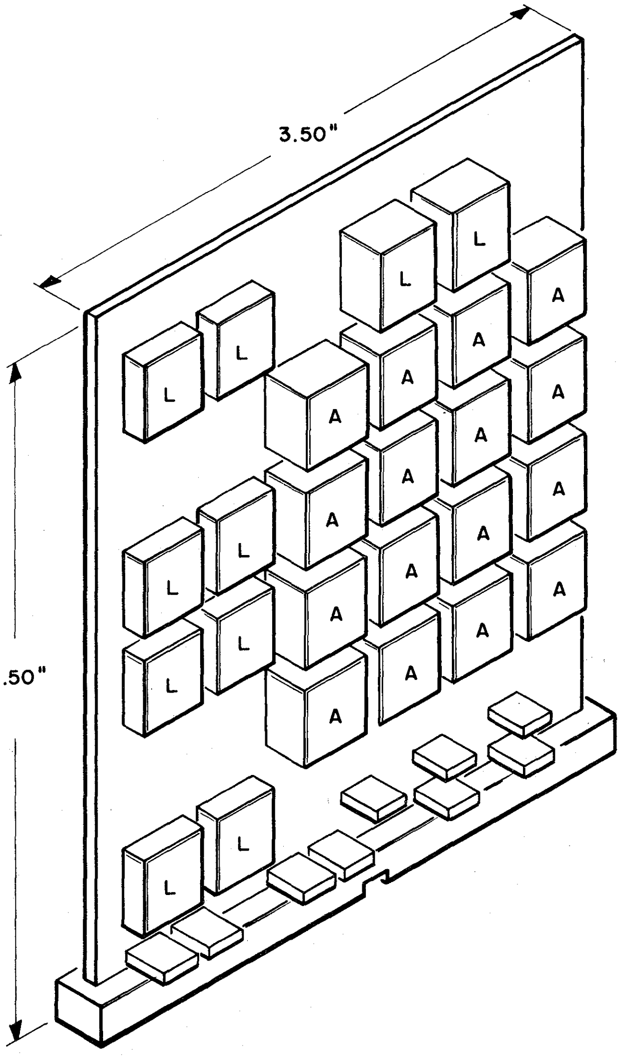


Figure 3—Storage card assembly

To keep the number of interconnections per module down to an acceptable level, the chips are arranged so that address, phase and voltage connections are shared. Input-output and select control lines are separate, resulting in a total of 28 active module connections.

Card design

Further assembly of the memory proceeds by placing 16 of the modules described above on a card as shown in Figure 3. This package or card also contains partial address decode, refresh, control logic, and address interface and phase drivers to provide fan-out for the memory modules. The card or memory subunit is actually a self-sufficient memory in its own right (except for logic and driving circuits), providing a storage capacity of

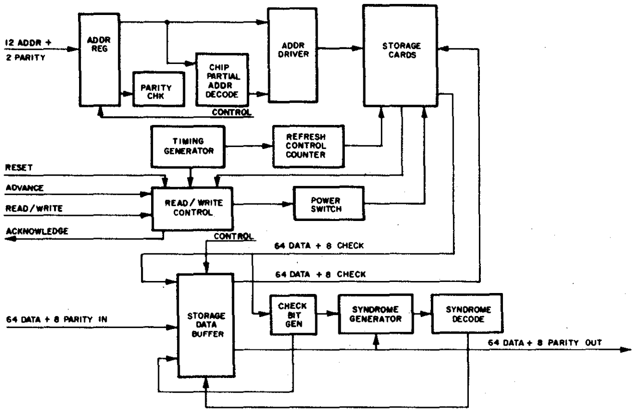


Figure 4—Functional diagram

10^6 bits. The 10^8 bit CCD memory is realized by stacking up the appropriate number of these building blocks. They lend themselves to several different memory organizations, the exact number depending on the organization chosen. In the "bit-per-card" design, a memory word consists of 64 data bits, 8 check bits, plus a position synchronization bit, for a total of 73 cards. The memory is operated in a mode where only one chip per card is selected at a time, providing a substantial savings in card power dissipation. Since the cards are independent sub-memories, serviceability is enhanced.

Summary of chip, module and card designs

Many of the reasons for the particular choice of chip, module and card design have been given in the preceding paragraphs. For further discussion, however, it is necessary to know how the memory will be organized, and this requires a knowledge of the electrical circuitry, packaging and error correction coding needed to implement the possible organizations. Discussion of these topics, comparisons and tradeoffs, however, will be based on the above described chip, a module with two chips and a card with 16 storage modules, logic and interface drivers as described. The storage array card is organized in 1024K word by 1-bit configuration.

TABLE III—Power Requirements For System Control Functions

FUNCTION	EQUIVALENT GATE COUNT	POWER (mw)
Address register	105	3920
Parity check circuit	56	640
Chip partial address	21	637
Address drivers	57	2570
Timing generator	43	2310
Refresh control counter	188	4030
Read/write control	161	6821
Power switch	147	7120
Storage data buffer	588	9500
Check bit generator	845	18120
Syndrome* generator	76	1440
Syndrome decode	180	3240
	2467	60.347 watts

* Syndrome-encoded signals generated as a result of a bit error from which the incorrect bit can be located.

Electrical and mechanical design

Circuit requirements

The circuits designed to implement the functions defined in the memory system design are tentative and have not been optimized. They are only intended to represent a reasonable estimate of complexity and power consumption. For the sake of discussion, all the numbers that follow pertain to the bit-per-card organization.

TABLE IV—Power Requirements (Watts) For Memory System

System logic	60.5
Storage array 73 cards @ 2.185	~160.0
Storage fan-out drivers	15.7
	236.2

The general functional diagram of the system is shown in Figure 4. Twelve address lines provide addressing to each of the 4096 2048-byte blocks of stored data. An eight-byte (64 data bit) parallel interface data bus provides a high data transfer rate with an acceptable investment in power and error correction circuitry. The power and equivalent gate count for each function are presented in Table III. The system power is presented in Table IV.

Packaging design relies upon IBM's standard card and board concepts. The technical objectives of this design concept are (1) minimization of interconnection complexities, (2) adequate environmental protection, (3) sufficient thermal efficiencies to employ forced ambient air cooling, (4) modular flexibility features suitable for expansion, and (5) favorable economic costs.

The overall packaging configuration of the memory system is pictured in Figure 5, resulting in a volume of 2.9 cubic feet.

The unit consists of a gate-structure containing four multilayer circuit boards, covers, interconnecting cables and pluggable modular card subassemblies containing logic and memory circuits. This unit design does not contain power supplies. It is assumed power and cooling air will be furnished by the host machine.

Power consideration

The CCD cell uses a dynamic charge storage principle and must be periodically regenerated to account for the charge "lost" from the storage node. This regeneration is accomplished by means of restore amplifiers in each shift register. Information is stored as charge in the potential wells, and the ability to differentiate between the amount of charge represents a bit of information. The minimum operating frequency of the shift registers is determined by detectable charge difference, which is a function of time, transfer efficiency and leakage rate. Careful consideration of these factors results in a lower shift rate of 5 kHz with a restore amplifier located every 64 bits. The restore amplifier senses the charge difference remaining at the end of a register and restores the charge levels to those corresponding to an initial one or zero for the next register segment. At this frequency, the chip requires a "standby" power of 19.4 mW. In the standby state, information integrity is maintained but all functions not required to that end are de-powered using pulse power techniques.

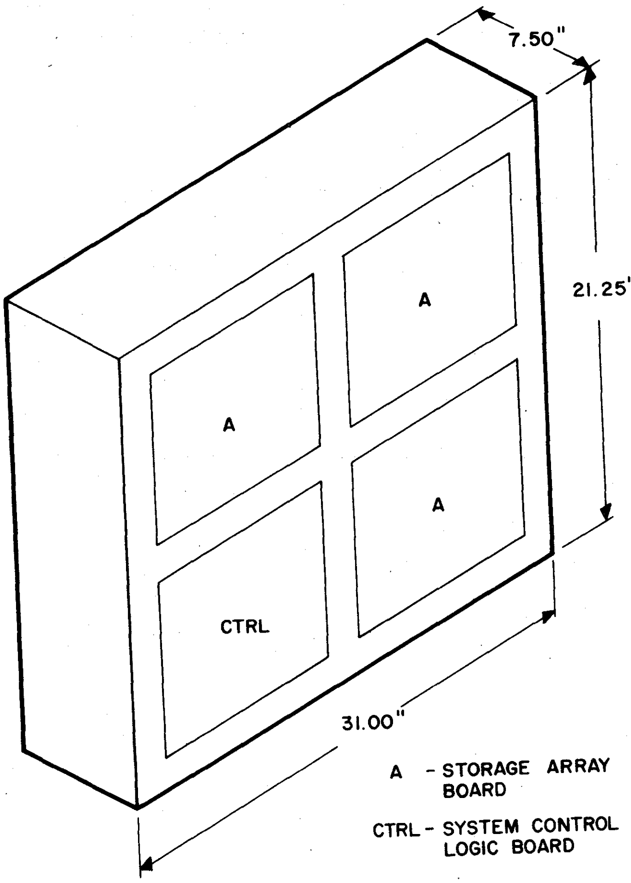


Figure 5—Eight megabyte storage unit

The maximum operating frequency is determined by the allowable power dissipation of the chip for forced air cooling. These considerations result in a shift frequency of 500 kHz with a selected chip (i.e., all circuits fully powered) power of 321 mW.

Error detection and correction

A preparatory phase of formulating a reliable storage system requires careful consideration of failure modes for the devices used in the system. Prediction of classes of device failures for the complete memory system is used to impose constraints upon both the chip and system organizations to assure the desired reliability. These considerations are based primarily upon the relative amount of circuitry used to implement the reliability-enhancement features.

To overcome the effects of shift register malfunctions, the memory system is organized so that each bit from a shift register is part of a word located on a different chip.[1] In this memory system organization, the memory device failure modes manifest themselves as either single or, with lessor probability, double bit errors.

The widely-used Hamming-type SEC/DEC* codes, for example, can correct any single-bit error in a memory word, but can only produce an error message in case of a double or higher-order bit error. Such codes, therefore, are most effective in systems organized so that as many failure mechanisms as possible cause only single-bit errors.

The design philosophy leading to the final, recommended, bit-per-chip system was as follows:

1. Offer random access to a block of 256 words of 64 bits each, and serial access to a specific word within that block.
2. Use SEC/DED codes to enhance reliability.
3. Achieve high performance operating characteristics at reduced power by supplying the high speed shift field to only the chips containing the desired shift registers, since only 64 shift registers (representing 0.2 percent of the total memory capacity) are accessed at any one time. This is accomplished by including phase gating logic on the chip, to be activated only when the chip is selected. Additional support electronics are

* Single-Error-Correction/Double-Error-Detection.

TABLE V—Summary of Chip Design Features

CONCEPTUAL DESIGN

- Self-contained CCD chip with on-chip decoding
- n-type substrate SAG technology
- Closed loop shift registers
- Operating speed of 5 to 500 kHz
- 64-bit shift register segments
- Shift register turnaround and restore amplifiers
- Four-phase operation
- 2.9×10^6 bits/in^2
- Operation in a machine environment

OPERATING FEASIBILITY MODEL

High Density Model

- n-type substrate SAG technology
- 256-bit single shift register segment
- 2.1×10^6 bits/in^2
- Operating speed of 500 Hz to 5 MHz
- Four-phase operation

Operating Memory Model

- n-type substrate technology
- 480-bit, 10 segment shift registers
- Shift register turnaround and restore amplifier
- Four-phase operation
- Operation in a machine environment
- Operating speed of 500 kHz
- Wide operating parameters
- I/O amplifier

necessary to insure data retention and regeneration; however, dc driver power dissipation is reduced.

FEASIBILITY MODELS

This section deals with the feasibility models that were built to test and demonstrate the major operating features of the conceptual memory design described above. The models were to be limited in capacity to that necessary to prove feasibility of the approach used in the conceptual design. The essential features of the conceptual chip design and the models are summarized in Table V.

The feasibility model chips differ from the conceptual chip not only in scale but also in that they do not demonstrate on-chip decoding or closed-loop operation. These have been relatively simple functions to accomplish, as shown by workers in this laboratory and industry as a whole. Mounting of the conceptual chip uses techniques well-known to industry.

Critical features of the conceptual design have been demonstrated by the feasibility models. These include the storage cell density, operating speed, sensing and amplification, and sufficient operating parameter tolerance for actual machine environment.

Device structure

The charge-coupled device uses basically MOS technology. The described structure consists essentially of three layers and is a junctionless device except for small diffused junctions that serve as input and output nodes of the shift register. The surface of a semiconductor, such as silicon, is oxidized to form a thin insulating layer. A metal pattern of electrodes is deposited on top of the insulator. In operation, the shift register depends on the transfer of charge from the potential well developed under one electrode to another by application of suitable voltages to these electrodes.

A cross-sectional view of the overlapped electrode devices fabricated in this test chip is illustrated in Figure 6. In the structure shown, each spatial bit has associated with it four independent electrodes. The Ø1, Ø3 electrodes, doped polysilicon, define the storage potential well (node) locations. The Ø2, Ø4 aluminum electrodes serve as transfer/isolation gates between storage nodes.

Device size

The proposed CCD storage cell is designed with an electrode separation of 0.15 mils and 0.05-mil overlap.

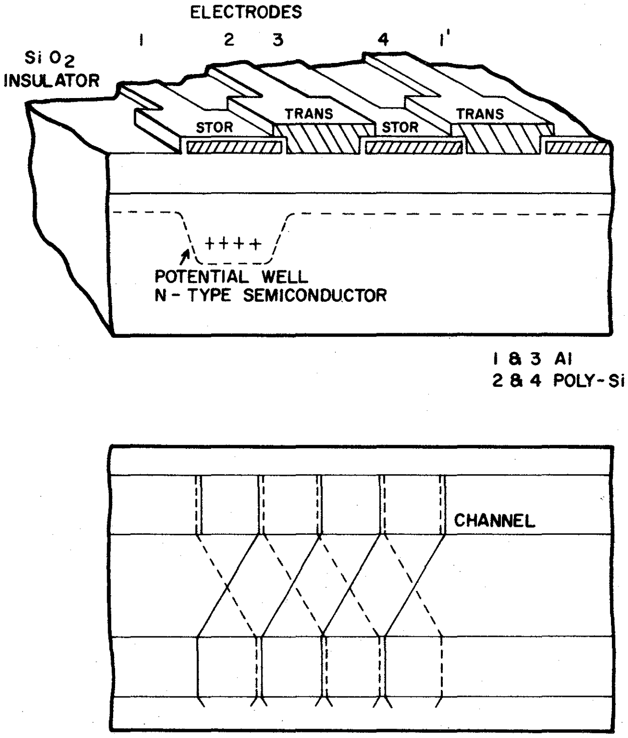


Figure 6—CCD storage cell

Channel width is 0.15 mils with a channel separation of 0.28 mils, resulting in an average area per bit of 0.35 mil.[2] In the high density devices described here,[2] the average area per bit is 0.48 mil[2] corresponding to a channel width and separation of 0.2 and 0.4 mils. Calculations of the potential barrier between channels indicate that the channel separation can be reduced to 0.2 mil, while stilll providing adequate isolation in 1-2 ohm-cm material.

Frequency response

In those pulse powered memory applications where access time minimization and power are important considerations, the frequency response is an important parameter. The frequency response curve for the normalized worst case one/zero difference (ΔD) of a typical 128-bit shift register is presented in Figure 7. Since the one/zero difference is directly related to the charge transfer efficiency, the data imply that the transfer efficiency characterizes device operation over an extremely wide clock frequency range.

The primary mechanism determining the low frequency limit of device operation is the thermal charge generation rate and the tendency of the empty potential wells to fill with thermally generated minority carrier charge. The 500-Hz data point indicates that such room temperature leakage is negligible at information dwell times of at least ¼ second.

The primary parameter affecting the high frequency limit of device operation is the surface mobility and its determination of the maximum transfer times needed to preserve efficient charge transfer between storage nodes. At a clock frequency of 5 MHz, the nominal transfer time duration is 60 nsec at which a slight drop in signal occurs. The signal difference is sufficiently large so that sensing is not impaired.

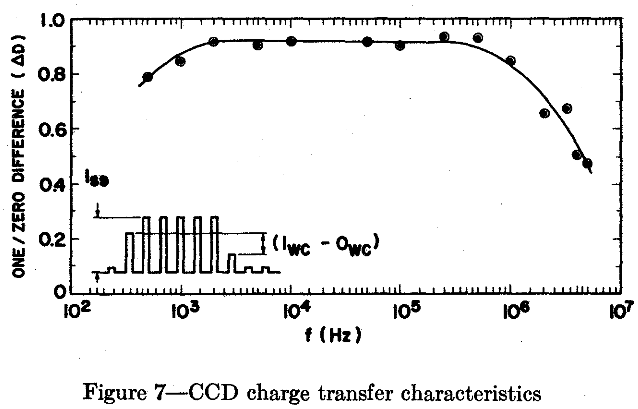


Figure 7—CCD charge transfer characteristics

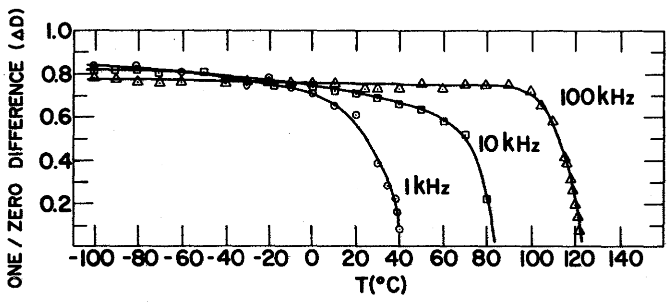
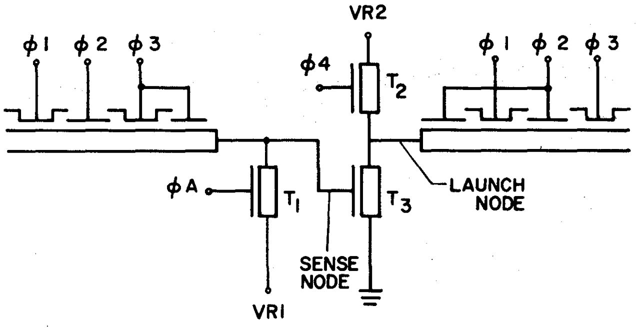


Figure 8—CCD temperature characteristics

There are two primary ways in which temperature can impact CCD operation. First, high frequency response is expected to be lowered with increasing temperature due to a decrease in surface mobility and a consequent decrease in charge transfer efficiency. Secondly, the low frequency response limit is expected to increase with increasing temperature due to the increased rate of thermally generated charge filling an empty potential well. The observed temperature dependence at the three clock frequencies shown in Figure 8 clearly displays the second effect described above. These data were obtained with the substrate biased at a relatively high level (8V), representative of stress conditions. At more moderate substrate bias potentials (~2V) the curves shown shift to significantly higher temperatures. This is due to the fact that a reduction in substrate bias reduces depletion region depth, resulting in a reduction in the rate of carriers filling the potential wells.

Operational memory

All the proposed circuit design concepts were exercised in a fully operational memory system.[3] The chosen memory system used dual 2880-bit shift register memory buffers operating at a fixed clock frequency of 500 kHz. The two 2880-bit shift register buffers were fabricated from six dual 480-bit open-loop shift register memory chips, serially connected to form the memory buffers. Each chip contains two 480-bit shift registers and is fabricated as previously described. Data flow proceeds in one direction for 47 ½ bits and is then amplified and launched in the reverse direction by a sensitive but simple amplifier that introduces an additional ½-bit delay. The restore amplifier, designed to operate in so-called fat-zero mode, consists of three FET's as illustrated in Figure 9. The amplifier inverts the signal and provides a small signal gain of about 80 from channel to channel. In operation, the signal at the launch



$C_{\text{sense node}}$	≅	0.06025 pf
$C_{\text{storage node}}$	≅	0.12 pf
$C_{\text{launch-to-sense}}$	≅	0.0015 pf

Figure 9—Restore amplifier

node is clamped to either VR_2 or near ground and the gain realized is only that necessary to compensate for the losses in the channels. Input and output support circuitry for the CCD shift registers interface directly to machine logic levels. Machine logic is diode-transistor with logic zero at ground and logic one at plus six volts. The chip output is capable of driving a minimum of one logic load (sink 1.7 mA to ground) and is fully compatible with both machine logic levels and CCD memory chips.

In summary, the work done in fabricating, testing, and designing the feasibility chips has demonstrated that the CCD technology is sufficiently mature and understood so that design and fabrication of a 10^8-bit storage system is possible with an acceptable risk factor.

REFERENCES

1 M Y HSIAO
A class of optimal minimum odd-weight-column SEC-DED codes
IBM Journal of Research and Development Vol 14 No 4 July 1970

2 N A PATRIN
Performance of very high density CCD structures
1972 Device Research Conference University of Alberta Edmonton Canada June 21-24 1972

3 N G VOGL T V HARROUN
Operating memory system using charge-coupled devices
1972 IEEE International Solid-State Circuits Conference February 16-18 1972

Josephson tunneling devices—A new technology with potential for high-performance computers

by W. ANACKER

IBM Corporation
Yorktown Heights, New York

INTRODUCTION

Superconducting thin film devices which exploit the Josephson effect and Giaever type tunneling have been observed to switch very fast while dissipating extremely little power. They show, therefore, good promise for high-performance computer circuits since the low power dissipation permits dense packaging with attendant short intercircuit signal delays, a prerequisite for effective utilization of fast switching circuits.

The devices are based on two discoveries of the early 60's. B. Josephson[1] predicted in 1962 that supercurrent can flow through nonsuperconducting and even insulating layers without causing voltage drops if these layers are thin enough. I. Giaever[2] discovered in 1960 that the tunnel characteristics of metal-oxide-metal sandwiches changes markedly when the metal electrodes become superconducting.

The possibility of building superconductive computers has been of interest ever since D. Buck[3] reported in 1956 on the operation of his "cryotron" switching device and demonstrated that this device is capable of performing logic and memory functions.

Efforts in several laboratories over the years to bring the cryotron technology to fruition have, however, met with failure. The main reason for the lack of success was that semiconductor technology has proved itself to be not only competitive but superior in switching speed and adaptability to large scale integration, facts which made it illogical to try to overcome technological problems of the cryotron technology which were unsolved at the time.

The basic difference between cryotron and Josephson tunneling circuits is that the latter have been shown to be orders of magnitude faster in switching operations than the former. In fact, it appears to date that Josephson tunneling devices have good potential to surpass semiconductor circuitry with respect to switching speed and system performance since packaging limitations due to heat removal problems already encountered in advanced semiconductor circuit networks are expected to be absent in Josephson tunneling circuit networks.

The basic effects underlying this technology are sketched in the next section before specific and preferred structures of switching gates, and their characteristics are discussed in the third section, and configurations and operations of memory and logic circuits in the fourth section. A review of presently known technological aspects is given in the fifth section followed by a brief summary in the final section.

SUPERCONDUCTIVITY, TUNNELING AND JOSEPHSON EFFECT

Superconductors are well-known to exhibit specific properties when they are cooled below a critical temperature T_c. Their electrical resistance drops to zero, and they behave like diamagnetic bodies, expelling magnetic fields. If they are arranged in closed loops, they can trap magnetic flux and sustain circulating persistent electrical current indefinitely.

These properties can be understood by postulating that superconductors comprise two interpenetrating electron fluids, a normal fluid of density n_n and a superfluid of density n_s. The normal fluid consists of single conduction electrons and the superfluid is made up of bound electron (Cooper) pairs which are the product of a condensation process taking place below the critical temperature and involving single conduction electrons with equal and opposite momentum and spin. The participating electrons occupy energy levels just below and above the Fermi level of the metals. The depletion

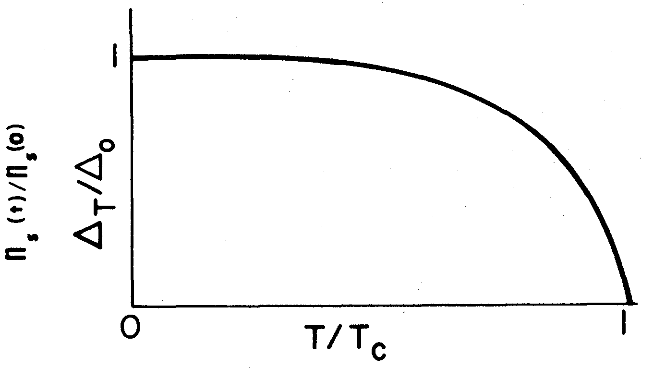


Figure 1—Normalized diagram of a superconducting energy gap Δ_T and superfluid $n_s(t)$ dependence on temperature T. Δ_o and $n_s(o)$ denote values at $T = 0$.[13]

of these levels leads to the development of a forbidden energy band, the so-called superconducting energy gap 2Δ (on the order of meV) around the Fermi level.

The Cooper pair system occupies a single energy state which lies below the Fermi energy by an amount of energy Δ (per electron) corresponding to the binding energy of the Cooper pairs in accordance with the classic (BCS) theory of superconductivity[4] by Bardeen, Schriefer and Cooper. The Cooper pair system is highly correlated throughout the superconducting region which is commonly expressed as the long range order of the superconducting state. The system can be described in quantum mechanical terms by a wavefunction $\psi(r)$, the phases $\phi(r)$ of which are related throughout the entire superconductor.

The Cooper pair system density n_s and the energy gap Δ are related to each other and depend on temperature as is shown in Figure 1. n_s and Δ are zero for $T > T_c$, rise first rapidly, then much more slowly with decreasing temperature until Δ is full developed and only the superfluid n_s exists at $T = 0$°K.

The second effect to be reviewed is electron tunneling. It is well-known that electric currents can be transported through a metal-insulator-metal sandwich with an attendant voltage drop across the thin insulator by means of the tunneling process. This process is readily understood as a consequence of the quantum mechanical concept of assigning to electrons in metals wavefunctions of the form $e^{i.\mathbf{k}.\mathbf{r}}$ with $\mathbf{r}$ being a space variable and $\mathbf{k}$ being the wave vector. For $\mathbf{k}$ imaginary—which is the case outside of the metal surface—the wavefunction decays exponentially. Since the square of the amplitude of the wavefunction at any point in space is interpreted as the probability of finding an electron at that point, it becomes apparent that electrons can be found with finite probability outside of a metal surface. These electrons can be captured by a second metal with available allowed energy states if the second metal is placed close enough to the first metal surface. The distance between the metals is not to exceed about 50 Å since the probability of finding electrons decreases rapidly with distance from the metal surface and the tunnel currents become extremely small.

Holm[5] has calculated the electron flux based on a simple one-dimensional model, from which the tunnel resistance R_{NN} of a tunnel junction can be deduced for $V \ll \phi_w$ as:

$$\frac{1}{AR_{NN}} = \frac{e^2}{h^2 d} (2m\phi_w)^{1/2} \exp - \frac{4\pi t}{h} (2m\phi_w)^{1/2} \quad (1)$$

with t and ϕ_w denoting the potential barrier thickness and height, respectively, A the junction area, e and m the electronic charge and mass, respectively and h Plank's constant.

Expression (1) indicates the exponential dependence of the tunnel resistance on the potential barrier height $\phi_w^{1/2}$ and thickness t and also that the voltage-current relation is linear.

The latter characteristic was shown by I. Giaever[6] to change when the electrodes become superconducting. He found that for voltages $V < 2\Delta/e$ the tunnel current is suppressed, rises rapidly at $V = 2\Delta/e$, and approaches the linear R_{NN} asymptotically for $V \gg 2\Delta/e$.

The discovery of this effect represents a direct confirmation of the existence of the superconductive energy gap as postulated in the BCS theory and it has served as a valuable tool to probe into the electronic structure of superconductors. As an example, Figure 2 shows

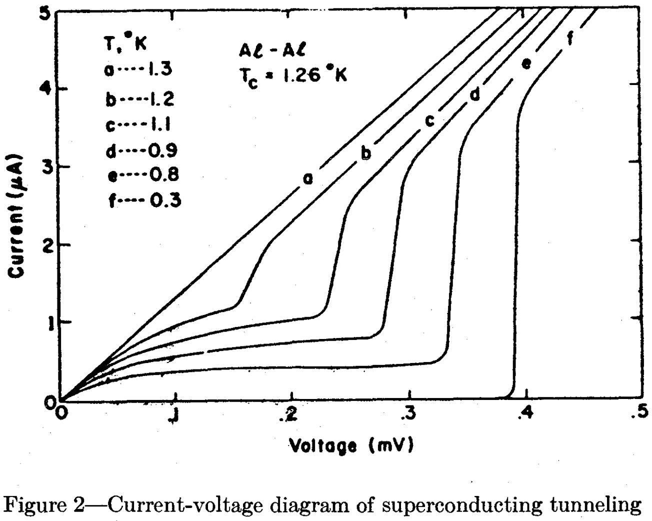


Figure 2—Current-voltage diagram of superconducting tunneling (with parameter a to f referring to decreasing temperature T).[22]

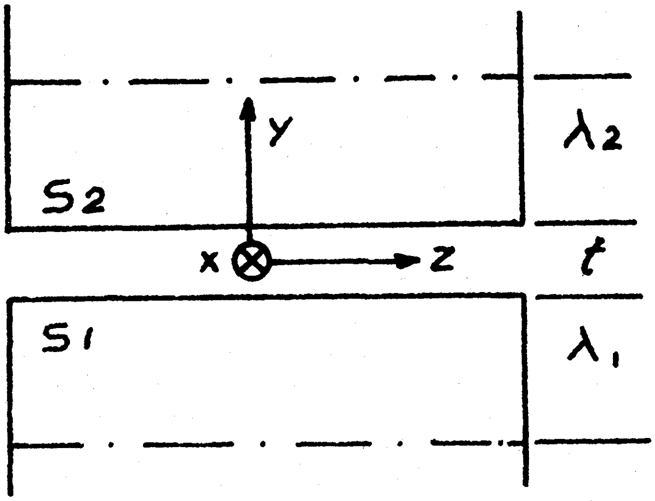


Figure 3—Geometry of superconductor-insulator-superconductor structure

by way of tunnel characteristics how the energy gap develops with decreasing temperature.

The third effect to be reviewed is the Josephson effect[7] itself. It is a property of the Cooper pair system and manifests itself in regions where superconductivity is weak, i.e., where the Cooper pair density is (locally) low. Figure 3 shows two superconductors S_1 and S_2 separated by a distance t and denotes a coordinate system in compliance with Equations 2, 3 and 4.

The Josephson effect[7] is contained in Equations 2, 3 and 4.

$$j(z,t)=j_1 \sin\phi \qquad (2)$$

$$\frac{d\phi}{dt}=\frac{2e}{\hbar}V \qquad (3)$$

$$\frac{d\phi}{dz}=\frac{2ed}{\hbar c^2}H_x \qquad (4)$$

with $\phi=\phi_1-\phi_2$; $V=V_1-V_2$; and $d=\lambda_1+\lambda_2+t$ and $\hbar=h/2\pi$. ϕ_1 and ϕ_2 denote the phases of the wavefunction $\psi(r)$ in S_1 and S_2 respectively; V_1 and V_2 the electrical potential of S_1 and S_2 respectively and λ_1 and λ_2 the London penetration depths of S_1 and S_2, respectively. e, h and c denote the electronic charge, Plank's constant and the speed of light, respectively.

Equation (2) states that a *dc* Josephson current $j(z,t)\leq j_1$ can flow between S_1 and S_2 for $\phi=\text{const}$ which according to Equation (3) indicates that the voltage between S_1 and S_2 is zero. Equation (3) indicates, on the other hand, that ϕ changes in time if a constant voltage V is impressed between S_1 and S_2 which, in conjunction with Equation (2) states that application of a voltage between S_1 and S_2 generates an ac Josephson current with a frequency which is proportional to the applied dc voltage. Equations (4) and (2) indicate that $j(z)$ is modulated in amplitude and even in direction along the z-coordinate in the presence of static magnetic fields penetrating the weak superconducting region. This leads to nonuniform internal current distributions which reduce the external current carrying capability of the weak superconducting region thus permitting one to control the (external) dc Josephson current threshold by applying magnetic fields. The experimental demonstration of the magnetic field dependence of the dc Josephson current threshold confirmed conclusively the existence of the Josephson effect.[23]

The Josephson effect has been observed in a variety of configurations such as superconductor-normal metal-superconductor sandwiches, point contacts, micron size constrictions in thin films, whiskers, even wires pressed together to form point contacts. The most suitable configuration for computer switching circuits appears, however, to be the Josephson tunneling gate which is described in the next section. An excellent and exhaustive discussion of Josephson devices is given by Matisoo.[8]

JOSEPHSON TUNNELING GATES

Figure 4 depicts schematically a Josephson tunneling switching gate. Two superconducting thin films S_1 and S_2 are shown partially overlapping and separated from each other by an insulating layer of 10 Å to 30 Å thickness, preferably an oxide grown on the bottom film S_1. They are overlaid by a third superconducting film insulated from S_1 and S_2 and patterned as a control line C. The entire structure is deposited on top of a superconducting and insulated ground plane in order to minimize circuit inductances.

The tunnel junction length and width are denoted by L and W, respectively, the oxide thickness by t, the London penetration depths in S_1 and S_2 by λ_1 and λ_2, respectively. The tunnel resistance R_{NN} of this configuration could be calculated from Equation (1) if t and ϕ_w were known, it can also be measured quite accurately at $T<T_c$ at a voltage V (with $2\Delta/e<V\ll\phi_w/e$). (Measurements at $T<T_c$ are necessary to eliminate dominating effects of the electrode resistances.)

Once R_{NN} is known, the Josephson current density j_1 can be calculated from:

$$j_1=K(\pi\Delta/2eR_{NN}.L.W)\ \tanh\ (\Delta/2kT) \qquad (5)$$

with the Boltzman constant k, a correction factor K to include strong coupling effects[9] (e.g., $K=0.91$ for tin

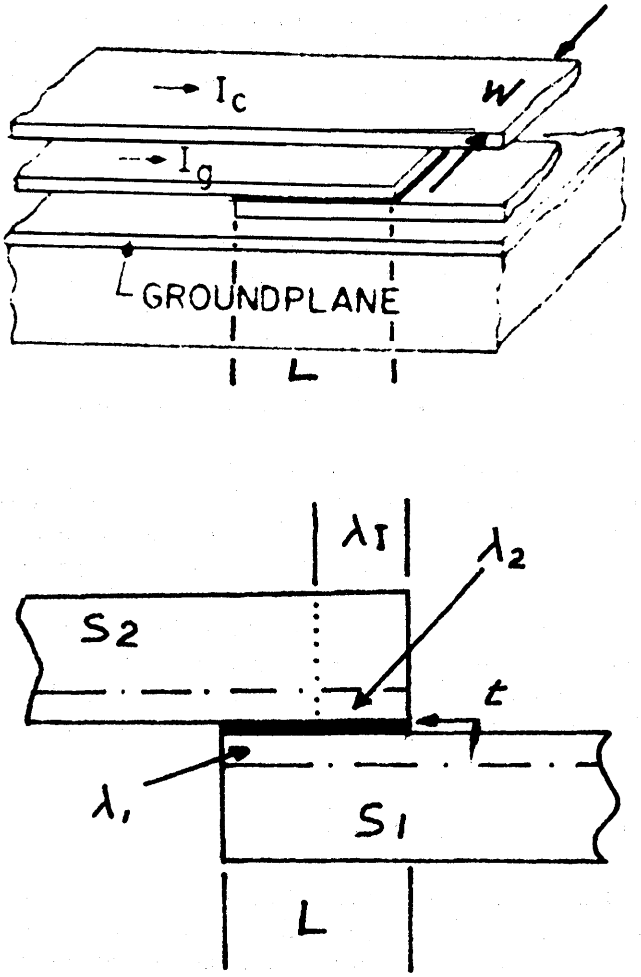


Figure 4—Josephson tunneling in-line gate configuration.[13]

and $K = 0.788$ for lead) and the other quantities as defined before. The temperature dependence of j_1 is mainly determined by $\Delta(T)$; thus the functional relationship of $\Delta(T)$ shown in Figure 2 holds to a very good approximation also for the Josephson current density j_1.

An important parameter for the control of Josephson current thresholds which is well defined in Josephson tunneling junctions is the Josephson penetration depth:

$$\lambda_J = (hc^2/8\pi e d j_1)^{1/2} \tag{6}$$

with $d = t + \lambda_1 + \lambda_2$, the junction "thickness" into which magnetic fields can penetrate, and all other quantities as defined before. The Josephson penetration depth λ_j denotes the distance, from the junction edges in which Josephson currents actually flow (as a consequence of Meissner effect) as shown in Figure 4.

Since $\lambda_J \propto j_1^{-1/2}$ and $j_1 \propto e^{-t}$, it is obvious that desirable current distributions can be readily obtained for a wide range of junction lengths L by simply changing the oxide thickness t.

The magnetic field dependence of the dc Josephson threshold current is strongly influenced by the current distribution in the junction.

Let us first consider the case of $\lambda_J \gg L$. Integration of Equation (4) with appropriate boundary conditions results in the expression:

$$I_{\max} = j_1 W \cdot L \frac{\sin \pi\Phi/\Phi_0}{\pi\Phi/\Phi_0} \cdot \sin \phi \tag{7}$$

with $\Phi_0 = hc/2e$, the superconducting flux quantum ($2.07 \cdot 10^{-15}$ V sec) and $\Phi = \mu s H \cdot L \cdot d$ the total magnetic flux penetrating the entire junction cross section for applied magnetic fields H. The dependence of $I_{\max}$ on Φ/Φ_0 is shown in Figure 5. The agreement between theory and experiment is excellent.

For junctions with nonuniform current distribution ($\lambda_j \simeq L$) the magnetic field dependence of the threshold current does not obey Equation (7); a full discussion of this fact is beyond the scope of this paper. The resultant dependence of the gate current threshold on the control current for a junction with $\lambda_j \sim (1/10)L$ is shown in Figure 6, displaying again the excellent agreement of the theoretical values (solid lines) and experimental data (dots). It should be noted that the slope of the line $c-a$ approaches unity for $\lambda_J/L \to 0$ while the slope of the line $c-d$ approaches infinity. The asymmetry with respect to the $I_{\max}$ axis is introduced in the so-called in-line configuration by the ground plane, which causes the magnetic fields of gate current and control current

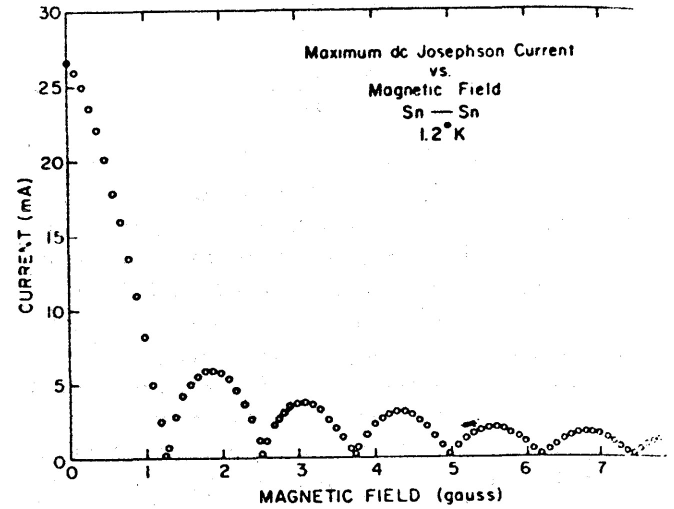


Figure 5—Normalized diagram of Josephson threshold current versus magnetic flux for junction with $\lambda_J \gg L$.[22]

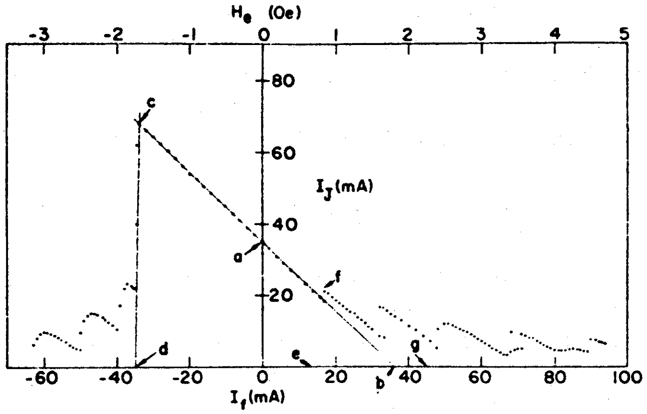


Figure 6—Diagram of Josephson gate current I_j versus control current I_F with $\lambda_J \ll L$.[8]

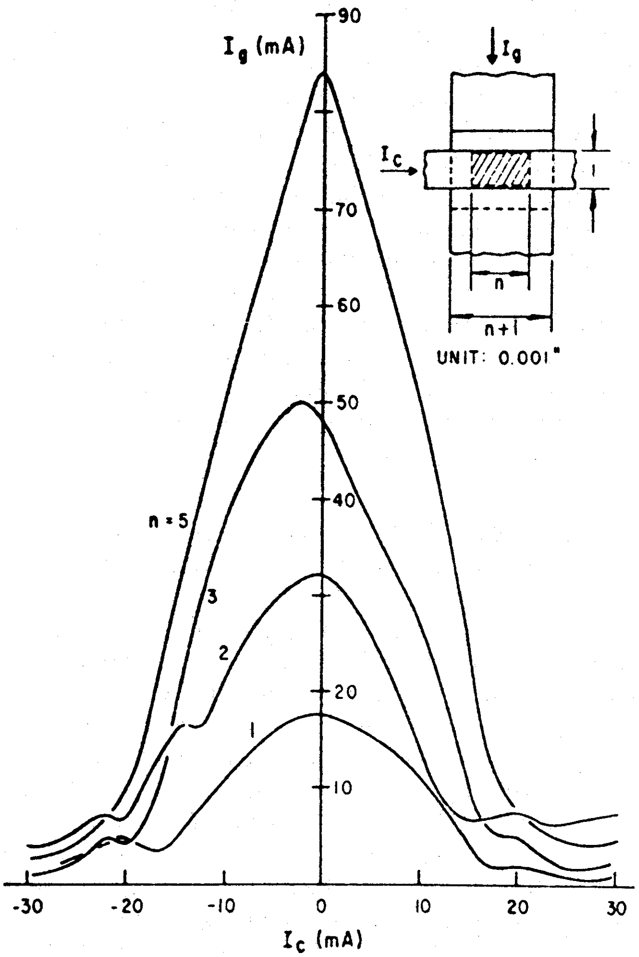


Figure 7—Diagram of Josephson gate current I_g versus control current I_c for window type crosscontrol gate configuration as shown on upper right (n denotes the ratio of gate line width to control line width). [10]

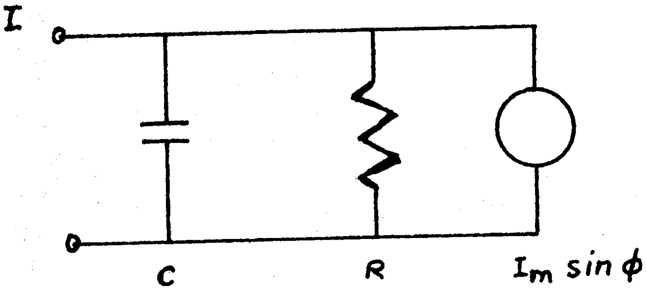


Figure 8—Equivalent circuit of Josephson tunneling junction, driven by current I

to add if currents flow in parallel and to subtract if currents flow antiparallel.

Symmetry with respect to the $I_{\max}$ axis can be restored by arranging the control line perpendicular to the gate lines in the so-called cross-controlled gate configuration. The $I_{\max}$ vs I_c characteristic of a window type version of this gate configuration as reported in Reference 10 is shown in Figure 7. The straight lines in Figure 7 indicate again a fairly nonuniform current distribution. It should be noted that this configuration permits one to raise the slope of the I_g vs I_c lines above unity by increasing the ratio $n = W_1/W_2$ which is frequently useful for gain considerations.

The dynamic behavior of Josephson tunneling gates

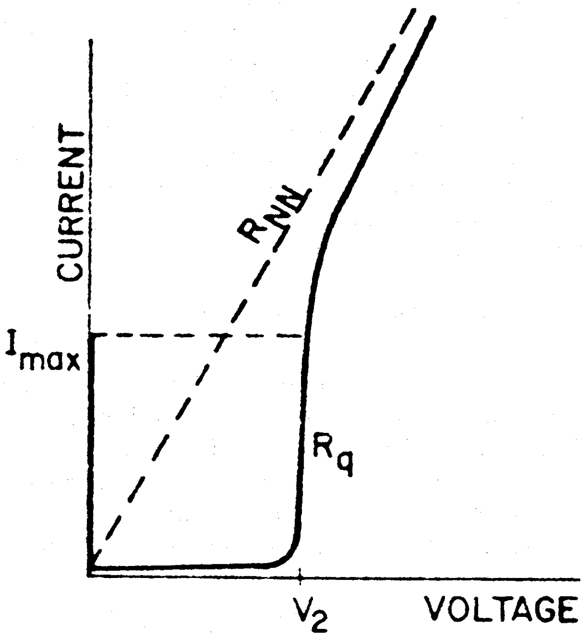


Figure 9—Current-voltage diagram of tunnel junction exhibiting Josephson and Giaever tunneling effects.[13]

is comprised of two aspects. One is the occurrence of hysteresis in the voltage-current curve, the other the dynamics during switching from $V=0$ to $V=2\Delta/e$ and back to $V=0$. Both effects are explained by resorting to the equivalent circuit of a Josephson tunneling junction[11] as shown in Figure 8. Two overlying metal films separated by a very thin dielectric layer evidently form a capacitance C, the tunnel resistance is expressed as a nonlinear resistance R (Giaever tunneling) and the Josephson effect is represented by a nonlinear Josephson current source ($I_{\min} \sin \phi$) with I_m being the maximum Josephson current through the junction. The differential equation of the equivalent circuit of Figure 8 in conjunction with Equation (3) produces the nonlinear equation:

$$C\frac{dv}{dt}+\frac{1}{R}V+I_{\min}\sin\phi=I \qquad (8)$$

Equation (8) has been solved numerically for linear resistance R[11] and a current source driving the junction with a dc current. The solutions show that hysteresis grows for increasing capacitance. Numerical solutions with nonlinear resistances R, more closely resembling Giaever type tunneling should reflect the experimental I-V curve of a Josephson tunneling junction as shown in Figure 9 rather closely. The hysteresis is of interest for circuit design since it permits the switching of gate currents in excess of control currents without requiring that the $I_{\max}$ vs I_c characteristic exceeds a slope of unity, thus providing for logic gain.

Equation (8) describes the time dependence of the gate voltage $V(t)$ when either the gate current I_g exceeds the Josephson current threshold $I_{\max}$ or the Josephson current threshold $I_{\max}$ is reduced below the gate current I_g (for example by application of a control current I_c). For the voltage rise, the Josephson current term ($I_{\max} \sin \phi$) can be neglected in Equation (8). The solution is then given by

$$V(t)=I_g\cdot R(1-e^{-t/RC}) \qquad (9)$$

with R being approximated by the average resistance of the Giaever tunnel curve for $0<V<2\Delta/e$. The voltage tends to rise to $V_1=I_g\cdot R$. It is, however, clipped at $V=2\Delta/e$ because of the small differential resistance of the Giaever tunnel curve at $V_g=2\Delta/e$. For $I_g\cdot R\gg V_g$ one may, therefore, express the rise time approximately by $\Delta t=I_g\cdot C/V_g$.

The ($I_{\max} \sin \phi$) term cannot be neglected in the switching operation from $V=V_g$ back to $V=0$. Equation (8) must be solved numerically in this case. Simulations indicate that the ac Josephson currents play a role in this case leading to oscillations and requiring careful device and current design.

The equivalent circuit of a Josephson tunneling gate driving a superconducting loop is derived by adding a parallel inductance to the equivalent circuit of Figure 8. The corresponding differential equation is given by:

$$C\frac{dv}{dt}+\frac{1}{R}V+\frac{1}{L}\int V\,dt+I_{\max}(\sin\phi)=I \qquad (10)$$

Equation (10) indicates oscillatory behavior even when the Josephson current term is neglected if care is not taken to provide sufficient circuit damping. The complete Equation (10) must be evaluated numerically for each case under consideration.

It is, of course, of interest to test the validity of the equivalent circuit models for Josephson tunneling circuits. Switching times for voltage rise and current steering were measured and compared with calculated switching times on the basis of the equivalent circuits of Figure 8 without and with inductances.[12] Agreement is very good for voltage risetimes as short as 65 psec and current steering time of 550 psec. The current and voltage levels in this experiment were 17 mA and 2.5 mV, respectively, leading to a switching energy of $1.4\cdot10^{-15}$ Joule per switching operation or about 10 μW in continuous operation with 50 percent duty cycle. The circuit dimensions in the experimental work were rather large; a junction with an area of about 20 mil^2 and a loop with a length of 410 mil were used. It is expected that miniaturization with attendant reduction in capacitances and inductances will result in even shorter voltage rise and current steering times and in lower power dissipation.

JOSEPHSON TUNNELING CIRCUITS FOR MEMORY AND LOGIC FUNCTIONS

Some features of cryotron technology such as infinite resistance ratios of the ON and OFF states, trapping of magnetic flux, persistent current, etc., will also be exploited in Josephson tunneling circuits. Specific features of Josephson tunneling gates, such as the I-V hysteresis, the well defined and relatively large gap voltage V_g and the ease with which I_g vs I_c characteristics can be designed will undoubtedly be used to advantage, too.

The basic circuit configuration for register and memory elements is a superconducting loop comprising Josephson tunneling gates for steering currents from one branch into another and changing the direction (or amount) of magnetic flux trapped in the loop. The storage of binary data can, for example, be accomplished by setting up persistent supercurrents which

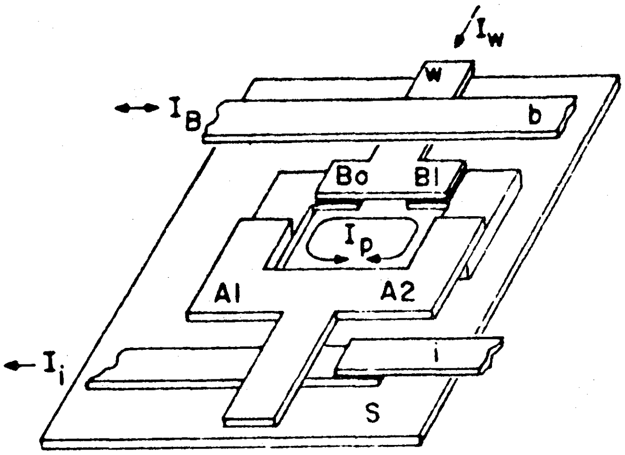


Figure 10—Memory cell comprising Josephson tunneling gates for coincidence selection and nondestructive readout.[13]

flow clockwise ("1") or counterclockwise ("0") as long as the loop remains superconducting.

As an example of a practical memory circuit[13] Figure 10 shows a superconducting loop comprising two Josephson tunneling gates for writing and controlling a third Josephson tunneling gate for nondestructive reading. The memory loop is in the selection line w; the control line b overlaying both write gates serves as a second selection line during writing; the interrogate line i, which comprises the sense gate S acts as a selection line during read out. The operation of the loop as a bit organized memory cell is described next under the assumption that there exists already a counterclockwise circulating current $I_w/2$ in the loop which represents a stored "0". First, a current pulse I_w is applied to line w, causing currents of amount $I_w/2$ to flow downwards through each branch of the loop. (Both branches are assumed to be equal inductance.) The applied and circulating currents superimpose, leaving the right branch without current and causing a total of I_w to flow in the left branch. The write gates are of the in-line configuration and their $I_{\max}$ vs I_c characteristic is assumed to be asymmetric ($\lambda_j < L$). A coincident current pulse I_B is then applied to the control line b. If I_B is directed from left to right (representing a write "0" signal) it increases $I_{\max}$ of the left write gate and decreases $I_{\max}$ of the right write gate. None of the gates will switch, however, since the actual gate currents—I_w in the left gate and 0 in the right gate—are smaller than the actual $I_{\max}$ values. If I_B is, however, directed from right to left (representing a write "1" signal), $I_{\max}$ of the left gate is decreased and $I_{\max}$ of the right gate is increased. With appropriate design $I_w > (I_{\max})_{\text{left}}$, the left gate switches and develops a finite voltage, initiating transfer of the current I_w to the right branch until the current through the left gate falls below $I_{\min} \sim 0$. This causes the left gate to return to $V=0$. The current I_w is now flowing through the right branch, and the loop has again become superconducting. Conservation of magnetic flux causes now a clockwise circulating current $I_w/2$ to flow in the loop representing a stored binary "1" upon termination of I_w in line w.

For reading of stored information, I_w is applied to w in coincidence with an interrogate current I_i flowing from right to left through line i. Superposition of I_w with a clockwise or counterclockwise circulating current subjects the sense gate underneath the right branch to a control current of $I_c = I_w$ and $I_c = 0$, respectively, thus rendering $I_{\max} < I_i$ in the sense gate in case of a stored "1", and $I_{\max} > I_i$ in case of a stored "0". In consequence, a finite voltage will be developed across the sense gate only for a stored "1" but not for a stored "0".

The current steering mode can be applied as well to larger loops comprising, for example, as branches selection lines w, b and i of memory arrays. It can be argued that array lines w, b and i in small memory arrays may be of about the same length as the loop (410 mil) of the high speed experiment mentioned in the third section. Hence, one may project memory cycle times in such arrays on the order of a nanosecond since usually the selection of array lines represents the major portion of the cycle time.

A drawback of the current steering mode in superconducting loops of extended length in high speed operation is the fact that the loops act as pairs of transmission lines above ground plane with a virtual short opposite the switching gate, thus causing multiple voltage and current reflections to travel along the lines and slowing the current steering operation. This behavior, especially undesirable in high speed logic circuits may be remedied. In Figure 11, the Josephson tunneling gate on the left drives a pair of striplines on top of a ground plane with a characteristic impedance Z_0. The strip lines which serve as control lines for other Josephson tunneling gates are terminated at the right

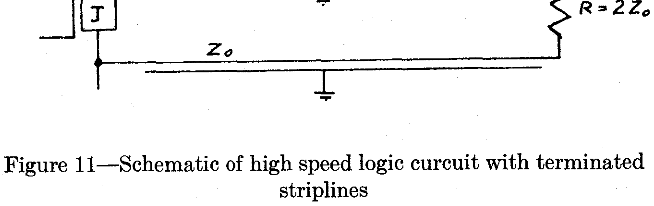


Figure 11—Schematic of high speed logic curcuit with terminated striplines

end by a resistor of resistance $R=2Z_0$. If the Josephson tunneling gate on the left switches from $V=0$ to $V=V_g$, two waveforms, one of amplitude $+V_g/2$ and the other of amplitude $-V_g/2$, will travel along the striplines; they will find a matched termination condition at the right end and be thus absorbed. Then the current $I_0=V_g/R$ will be established in the striplines right after the wavefront has propagated along the line. This current is a function of the (well defined) gap voltage V_g and the resistor R only.

Logic functions can be performed in different ways, for example, by overlaying several control lines on top of a Josephson tunneling junction or by interconnection of Josephson tunneling gates. If the 410 mil long loop in the high speed experiment had been properly terminated, the loop current would have been fully established in about 170 psec. (This assumes a 65 psec risetime and a phase velocity of 10^{10} cm/sec in the stripline.) In this case also, miniaturization is likely to reduce risetime and propagation delay.

TECHNOLOGICAL ASPECTS

An important question is, of course, whether useful devices and circuits can be made reproducibly, reliably and economically. A full assessment can only be made on the basis of a fairly extensive technological study; short of that, one must be content with collecting whatever data is available to date.

Following a recent review[14] it can be stated that Josephson tunneling junctions have been prepared using a variety of superconducting metals, tunnel barriers and preparation methods. A selection of superconductors, along with their critical temperatures T_c and energy gap 2Δ, which have reportedly been used for Josephson tunneling junctions are listed in Table I. Since operation at $T \leq \frac{1}{2}T_c$ is preferred because of the weak temperature dependence of Δ and j_1 in this range, niobium and lead electrodes are desirable candidates for this technology since they permit operation at about 4.2°K, the boiling point of liquid helium at atmospheric pressure.

TABLE I—Superconducting Electrodes for Josephson Tunneling Junctions

Electrodes	T_c (°K)	$2\Delta_o$ (meV)
Nb	9.2	2.9
Pb	7.2	2.5
Sn	3.7	1
In	3.4	1
Al	1.26	0.38

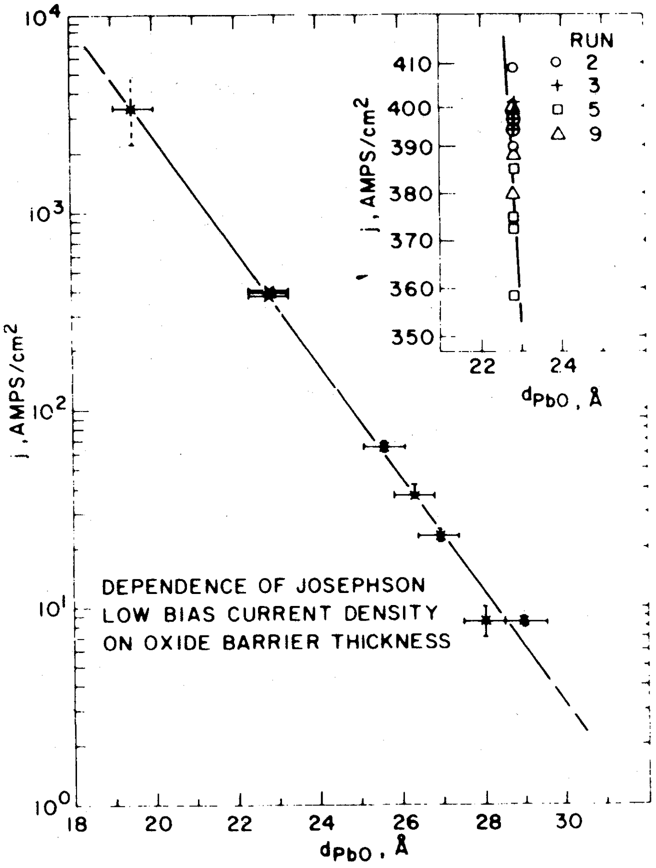


Figure 12—Diagram of Josephson current density versus oxide thickness in Pb-PbO$_x$-Pb junctions with indication (insert on upper right) of run to run reproducibility of barrier thickness and current density.[15]

Native oxides grown on the base electrode are preferable as suitable tunnel barriers for Josephson tunneling junctions. Successful attempts of barrier preparation by deposition of semiconductor films such as CdS,[15] tellurium[16] have also been reported. However, pinholes in the thin semiconductor films appear to pose a problem which is usually overcome by a subsequent oxidation step which fills the pinholes in the semiconductor films again by a native oxide. Hence, it appears to date, that native oxides grown on the bottom film electrode S_1 provide the best chance of getting shortfree tunnel barriers which are sufficiently thin to provide usable Josephson current densities for computer circuits.

Of major importance is the question of whether it is possible to control the growth of a thin oxide film with a thickness on the order of 20 Å to 30 Å sufficiently

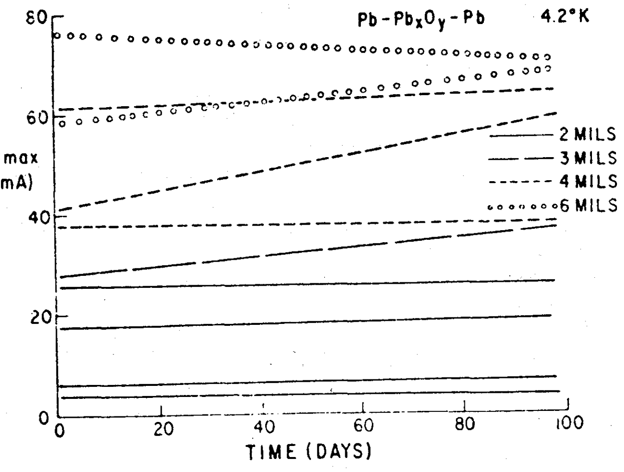


Figure 13—Diagram of Pb-PbO$_x$-Pb junction stability over extended period of time.[19]

uniformly and reproducibly to obtain reasonably reproducible Josephson current densities j_1. Figure 12 depicts the results of a thermal oxidation experiment[17] to produce Pb-PbO$_x$-Pb junctions in a well controlled oxygen atmosphere. The spread of current density j_1 is about ±6 percent between independent oxidation runs and even less on the same samples.

Dc-glow discharge experiments have been reported[18] to result in a logarithmic time dependence of the oxide thickness from which one can conclude that the reproducibility is reasonable here, too.

Another interesting variation of the dc glow discharge oxidation has been reported[19] in which an *rf* source is used rather than a dc supply and the sample is affixed to the cathode. Two competing processes, oxide growth and removal of oxide by *rf*-sputtering are apparently at work leading to a simultaneous process of surface cleaning and oxide formation. The inherent surface cleaning aspect makes this method attractive for large scale circuit integration where the electrodes are being subjected to photoresist process steps prior to the formation of the oxide tunnel barriers.

Another technological aspect concerns the stability of the junction characteristics when they are thermally cycled between cryogenic and room temperatures and during shelf life. It was found that niobium junctions[20] are excellent in this aspect. Figure 13 shows that lead-lead oxide-lead junctions[21] can also be kept for an extended time at room temperature and can be thermally cycled a number of times without undue change of characteristics.

To date, the technological issues related to Josephson tunneling circuits may be described as having yielded some encouraging results; they can, however, not yet be considered as being solved. More effort will have to be invested to assess the feasibility and bring the technology to fruition.

SUMMARY

Relevant basic physical and computer circuit aspects of Josephson devices have been reviewed. It has been shown that Josephson tunneling devices are well characterized and perform in very good agreement with numerical calculations on the basis of simple models. They are potentially useful for high performance computer applications since they possess the necessary attributes of extremely high switching properties, and of extremely little power dissipation; they adapt readily to miniaturized LSI fabrication processes.

They do not suffer from basic switching speed limitations of cryotron circuits caused by superconducting—normal conduction phase changes and low voltages. On the contrary, they promise to surpass semiconductor circuit networks in operational speed, and provide memory functions with zero standby power.

Some technological aspects appear favorable, however, much more work is required in order to demonstrate feasibility. The potential, however, appears to warrant effort to address and solve technological problems.

REFERENCES

1 B D JOSEPHSON
Possible new effects in superconductive tunneling
Physics Letters Vol 1 Page 251 1962

2 I GIAEVER
Energy gap in superconductors measured by electron tunneling
Physical Review Letters Vol 5 No 4 1960

3 D A BUCK
The cryotron—a superconductive computer component
Proceedings of the IRE April 1956

4 J BARDEEN L N COOPER J R SCHRIEFER
Theory of superconductivity
Physical Review Vol 108 No 5 1957

5 R HOLM
The electric tunnel effect across thin insulator films in contacts
Journal of Applied Physics Vol 22 No 5 1951

6 I GIAEVER K MEGERLE
Study of superconductors by electron tunneling
Physical Review Vol 122 May 1961

7 B D JOSEPHSON
Coupled superconductors
Advances in Physics Vol 14 Page 419 1965

8 J MATISOO
Josephson-type superconductive tunnel junctions and applications
IEEE Transactions on Magnetics Vol MAG-5 1969

9 T A FULTON D E McCUMBER
dc Josephson effect for strong-coupling superconductors
Physical Review Vol 175 No 2 1968

10 J P PRITCHARD JR W H SCHROEN
Superconductive tunneling device characteristics for array application
IEEE Transactions on Magnetics Vol MAG-4 No 3 1968

11 W C STUART
Current-voltage characteristics of Josephson junctions
Applied Physics Letters Vol 12 Page 277 1968 and
D E McCUMBER
Effect of ac impedance on dc voltage-current characteristics of superconductor weak-link junctions
Journal of Applied Physics Vol 39 Page 3113 1968

12 H H ZAPPE K R GREBE
Ultra-high-speed operation of Josephson tunneling devices
IBM Journal of Research and Development Vol 15 No 5 1971

13 W ANACKER
Potential of superconductive Josephson tunneling technology for ultrahigh performance memories and processors
IEEE Transactions on Magnetics Vol MAG-5 No 4 1969

14 J MATISOO
Josephson tunnel junctions
Conference Digest 1972 Applied Superconductivity Conference (to be published)

15 I GIAEVER
Photosensitive tunneling and superconductivity
Physical Review Letters Vol 20 No 23 1968

16 J SETO T VAN DUZER
Supercurrent tunneling junctions with tellurium barriers
Applied Physics Letters Vol 19 No 11 1971

17 J M ELDRIDGE J MATISOO
Measurement of tunnel current density in a metal-oxide-metal system as a function of oxide thickness
Proceedings of the 12th International Conference on Low Temperature Physics September 1970

18 J L MILES P H SMITH
The formation of metal oxide films using gaseous and solid electrolytes
Journal of Electrochemical Society Vol 110 Page 1240 1963

19 J H GREINER
Josephson tunneling barriers by rf sputter etching in an oxygen plasma
Journal of Applied Physics Vol 42 No 12 1971

20 L S HOEL W H KELLER J E NORDMAN A C SCOTT
Niobium superconductive tunnel diode integrated circuit arrays
Solid State Electronics (to be published)

21 W SCHROEN
Physics of preparation of Josephson barriers
Journal of Applied Physics Vol 39 No 6 1968

22 D N LANGENBERG D J SCALAPINO B N TAYLOR
Josephson-type superconducting tunnel junctions generators of microwave and submillimeter wave radiation
Proceedings of the IEEE Vol 54 No 4 1966

23 J M ROWELL
Magnetic field dependence of the Josephson tunnel current
Physical Review Letters Vol 11 No 5 1963

Magnetic bubble computer systems

by R. C. MINNICK

Rice University and Consultant to the Monsanto Company
Houston, Texas

P. T. BAILEY and R. M. SANDFORT

The Monsanto Company
St. Louis, Missouri

and

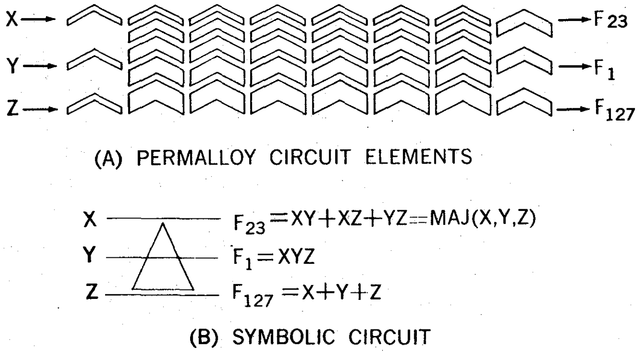
W. L. SEMON

Syracuse University and Consultant to the Monsanto Company
Syracuse, New York

INTRODUCTION

The purpose of this paper is to extend the work presented earlier[1] on the methods for performing logic on magnetic bubble chips, and to illustrate these techniques by presenting the outline of a design for a general-purpose digital computer.

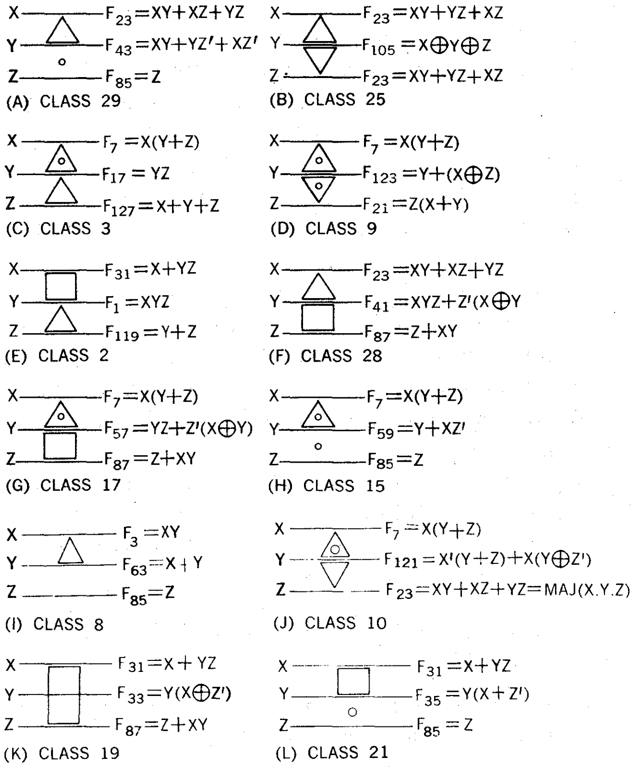
Magnetic bubble materials have received extensive treatment in the literature during the past few years,[2-9*] and numerous memories have been designed and tested



(A) PERMALLOY CIRCUIT ELEMENTS

(B) SYMBOLIC CIRCUIT

Figure 1—Primitive realization for class 1



(A) CLASS 29

(B) CLASS 25

(C) CLASS 3

(D) CLASS 9

(E) CLASS 2

(F) CLASS 28

(G) CLASS 17

(H) CLASS 15

(I) CLASS 8

(J) CLASS 10

(K) CLASS 19

(L) CLASS 21

Figure 2—Additional primitive realizations

[*] Only a few of the more important materials papers are cited.

	INPUT PERMUTATION																	
CLASS	1 XYZ			2 XZY			3 YXZ			4 YZX			5 ZXY			6 ZYX		
1	23	1	127	23	1	127	23	1	127	23	1	127	23	1	127	23	1	127
2	31	1	119	31	1	119	55	1	95	55	1	95	87	1	63	87	1	63
3	7	17	127	7	17	127	19	5	127	19	5	127	21	3	127	21	3	127
4	3	125	23	5	123	23	3	125	23	17	111	23	5	123	23	17	111	23
5	3	29	119	5	27	119	3	53	95	17	39	95	5	83	63	17	71	63
6	115	5	31	117	3	31	79	17	55	93	3	55	47	17	87	59	5	87
7	3	61	87	5	91	55	3	61	87	17	103	31	5	91	55	17	103	31
8	3	63	85	5	95	51	3	63	85	17	119	15	5	95	51	17	119	15
9	7	123	21	7	125	19	19	111	21	19	125	7	21	111	19	21	123	7
10	7	121	23	7	121	23	19	109	23	19	109	23	21	107	23	21	107	23
11	63	67	21	95	37	19	63	67	21	119	25	7	95	37	19	119	25	7
12	59	71	21	93	39	19	47	83	21	117	27	7	79	53	19	115	29	7
13	43	21	87	77	19	55	43	21	87	113	7	31	77	19	55	113	7	31
14	35	21	95	69	19	63	11	21	119	81	7	63	13	19	119	49	7	95
15	7	59	85	7	93	51	19	47	85	19	117	15	21	79	51	21	115	15
16	29	103	19	27	103	21	53	91	7	39	91	21	83	61	7	71	61	19
17	7	57	87	7	89	55	19	45	87	19	101	31	21	75	55	21	99	31
18	63	65	23	95	33	23	63	65	23	119	9	23	95	33	23	119	9	23
19	31	33	87	31	65	55	55	9	87	55	65	31	87	9	55	87	33	31
20	35	29	87	69	27	55	11	53	87	81	39	31	13	83	55	49	71	31
21	31	35	85	31	69	51	55	11	85	55	81	15	87	13	51	87	49	15
22	35	93	23	69	59	23	11	117	23	81	47	23	13	115	23	49	79	23
23	15	51	85	15	85	51	51	15	85	51	85	15	85	15	51	85	51	15
24	71	29	51	39	27	85	83	53	15	27	39	85	53	83	15	29	71	51
25	23	105	23	23	105	23	23	105	23	23	105	23	23	105	23	23	105	23
26	61	67	23	91	37	23	61	67	23	103	25	23	91	37	23	103	25	23
27	23	99	29	23	101	27	23	75	53	23	89	39	23	45	83	23	57	71
28	23	41	87	23	73	55	23	41	87	23	97	31	23	73	55	23	97	31
29	23	43	85	23	77	51	23	43	85	23	113	15	23	77	51	23	113	15
30	71	25	55	39	25	87	83	37	31	27	37	87	53	67	31	29	67	55
31	83	29	39	53	27	71	71	53	27	29	39	83	39	83	29	27	71	53

Figure 3—Table of the 3-3 circuits

using this medium. However, only a small amount of logic research has been reported for magnetic bubbles.[4–22] It is conjectured that the relatively slow data rates in bubble materials (10^5 to 10^7 bits per second in garnets) account for this: designers anticipate performing the memory functions (perhaps including the address decoders) on bubble chips, and the logical functions on silicon.

This approach, while it may be satisfactory in certain cases, appears to be inadequate if the number of data channels between the memory and the logic is large. This is because most acceptable bubble readout methods require a large area in order to obtain reasonable signal-to-noise ratios. Furthermore, by having logic separate from the memory, numerous opportunities are missed for useful intermixtures of the two. Therefore, it is expected that there will be a significant need for logic accomplished in the bubble medium.

SUMMARY OF CONSERVATIVE BUBBLE LOGIC

Magnetic bubble circuits that are memory-free and do not create or destroy bubbles are termed *conservative*,[20] and a conservative magnetic bubble circuit having n input variables and k output combinational switching functions of these variables is called an *n-k circuit*. The 2-2, 2-3, 3-3, 3-4, and 3-5 circuits have been treated earlier[1,20] and these results will be briefly summarized.

One of the 3-3 circuits is shown as Figure 1, first in terms of the permalloy circuit elements, and then in a symbolic form. It is observed in Figure 1(A) that a gradient is established in the field of the circuit elements because of the varying heights of the chevrons: a single bubble which enters on any track is moved downward to exit on the bottom track. However, if two bubbles enter on any pair of tracks, the mutual repulsion forces them to the top and bottom output tracks. The decimal subscripts on the output functions, F_{23}, F_1 and F_{127} indicate, when they are converted to binary, the truth-table form for the functions.

Twelve other 3-3 circuits are shown in symbolic form as Figure 2. In this figure, the rectangle represents a field of circuit elements with no gradient, the circled triangle is a strongly graded field of elements, and the small circle represents two bubble tracks sufficiently close together that bubble repulsion is effective.

There are 729 3-3 circuits;[1] for the convenience of presentation, these are arranged into 31 equivalence classes under all permutations of the three inputs and three outputs. These equivalence classes and the six corresponding input permutations are given as Figure 3. One member of each equivalence class is realized in terms of the thirteen primitive realizations in Figure 4. By permuting the inputs and outputs of the magnetic bubble circuits in Figure 4 in accord with the table of Figure 3, all 729 3-3 circuits may be formed.

NON-CONSERVATIVE BUBBLE LOGIC

It is possible to create a bubble with a device called a *generator*, to destroy one with an *annihilator* and to split a bubble on one track into a bubble on each of two tracks with a *splitter*. These devices, along with several others that are used in this paper, are symbolized as shown in Figure 5. If generators, annihilators, splitters or similar devices are used in bubble circuits, the circuits are termed *non-conservative*. In this section several non-conservative bubble circuits will be developed.

The class 24 conservative 3-3 bubble circuit in Figure 4 is of some interest, because it is a controlled permutation.[23] This circuit is drawn with a different output permutation in Figure 6(A), and symbolically in terms of the Kautz-Levitt-Waksman notation as Figure 6(B). Because this cell has useful logical properties, it is of some advantage to reduce the number of bubble-track crossings. By introducing two generators and four annihilators, it is possible to eliminate all such crossings for this cell; the resulting non-conservative circuit is shown as Figure 6(C). If one uses bubble splitters, an improved circuit can be found: it is shown as Figure 6(D).

A slight variation of the circuit of Figure 6(D) yields a circuit for performing a crossing of two bubble tracks without actually having them physically cross. This circuit is drawn as Figure 7(A). In Figure 7(B) is a conservative bubble circuit which converts a non-disjoint crossover into a disjoint one (see Figure 5). The advantage of this latter circuit is that the operating margins for a disjoint crossover are wider than for a non-disjoint one.

In order to obtain fan-out, some means for replicating a bubble is needed. A bubble splitter is a well-known example of such a replicator. A *logical bubble splitter* can be formed with a non-conservative bubble circuit as shown in Figure 8(A). If one has need for a larger number of bubbles, the bubble *multiplier* of Figure 8(B) can be used, or a tree of these can be assembled as shown in Figure 8(C). If this tree of multipliers has *n*

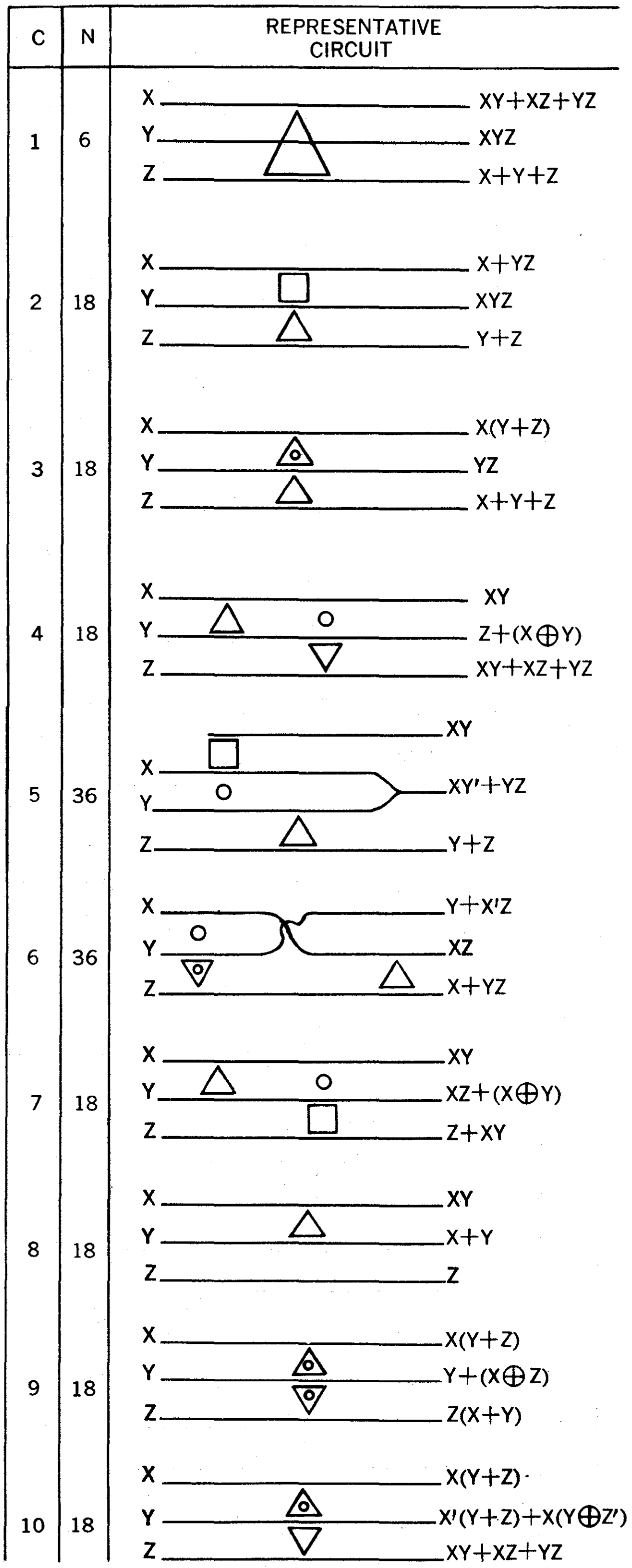


Figure 4—Realizations for all 3-3 circuits

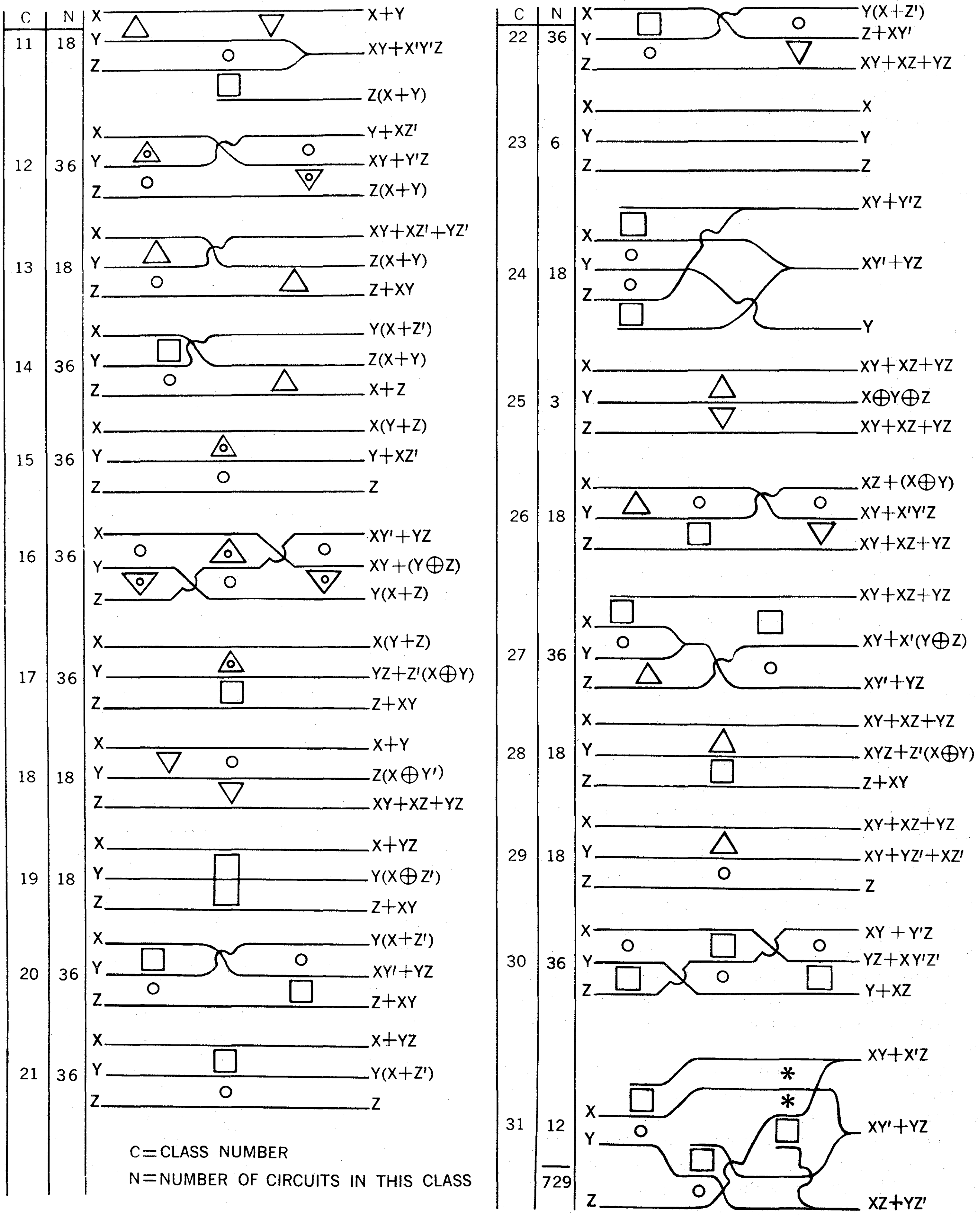


Figure 4—(continued)

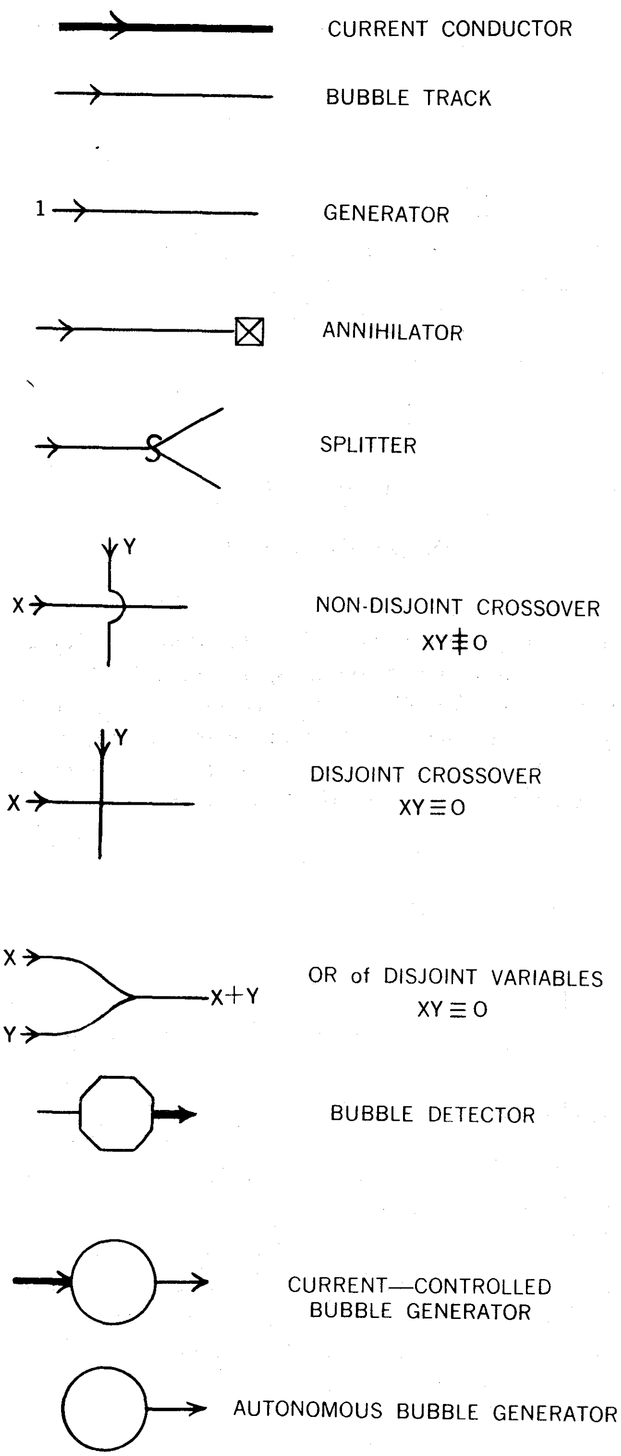
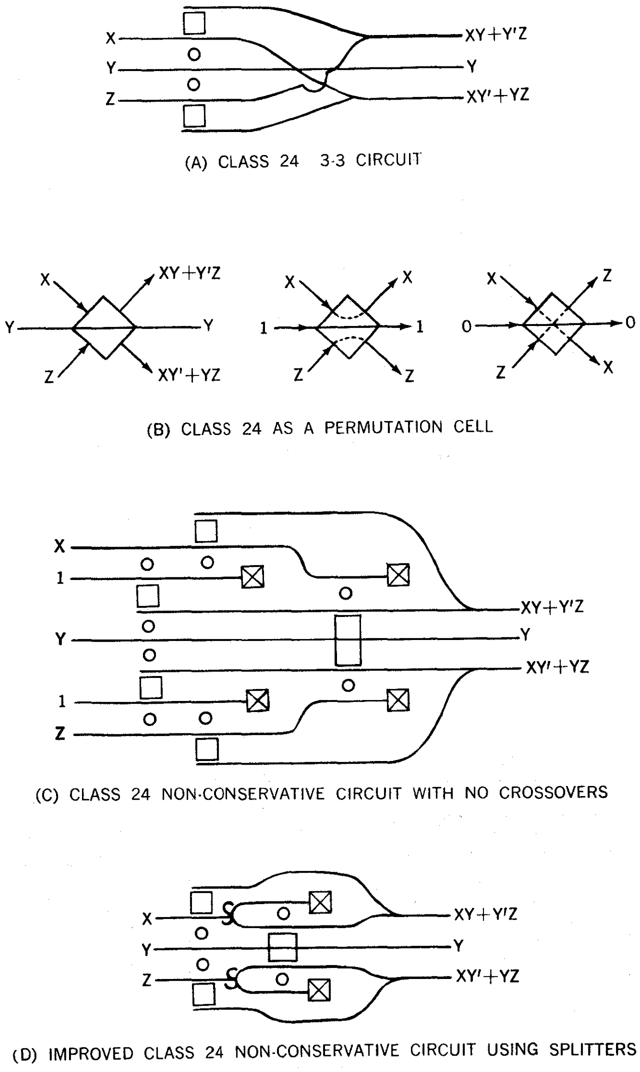


Figure 5—Symbols for bubble circuits



(A) CLASS 24 3-3 CIRCUIT

(B) CLASS 24 AS A PERMUTATION CELL

(C) CLASS 24 NON-CONSERVATIVE CIRCUIT WITH NO CROSSOVERS

(D) IMPROVED CLASS 24 NON-CONSERVATIVE CIRCUIT USING SPLITTERS

Figure 6—Non-conservative circuits for class 24

levels, there are $(5^n + 1)/2$ bubbles produced at the output for a single input bubble.

Bubble flip-flops are essential to any digital system, and both trigger (T) flip-flops and reset-set (R-S) flip-flops have been reported.[12] Both of these flip-flops depend on a trapped bubble in an idler loop. It is instructive, however, to display alternative flip-flops based on the 3-3 circuits of Figure 4. Two of these are shown as Figure 9. While these flip-flops are based on the circuit for equivalence class 9, several other circuits would do as well.

DECODERS

The bubble circuits that have been described can be assembled into decoders. For instance, Figure 10 shows

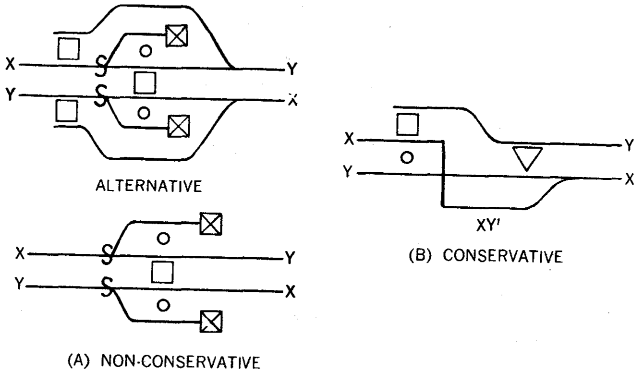


Figure 7—Logical crossover circuits

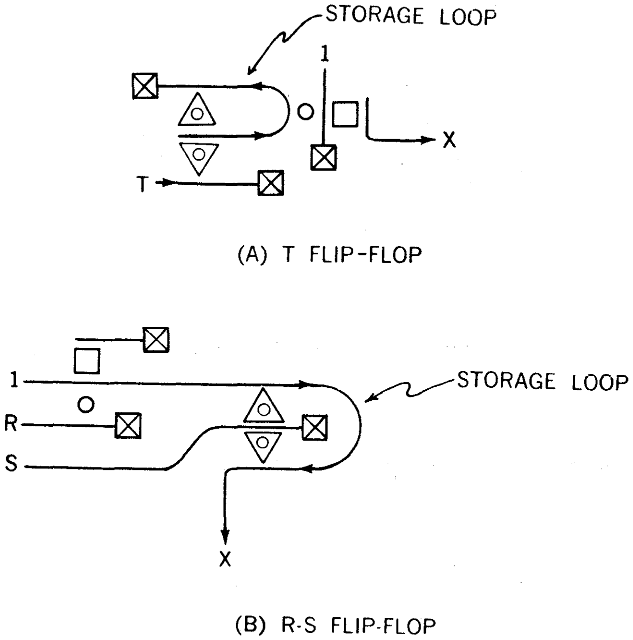


Figure 9—Bubble flip-flops

a four-variable decoder based on the circuit for class 21; it has 11 bubble-track crossings, one generator and four annihilators. An alternative version is shown as Figure 11; in this case the number of crossings is the same, while four splitters and four additional annihilators are inserted to make the layout simpler.

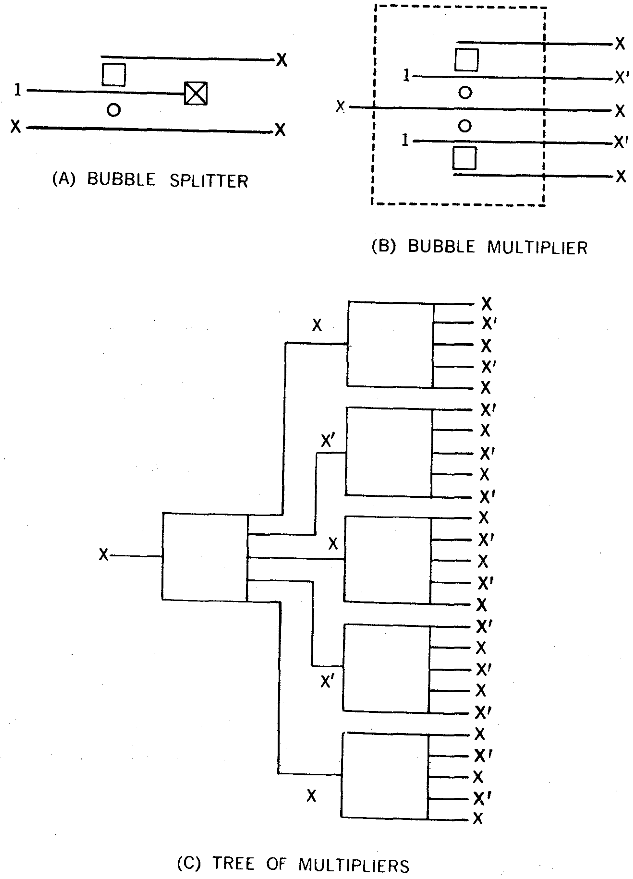


Figure 8—Logical bubble multipliers

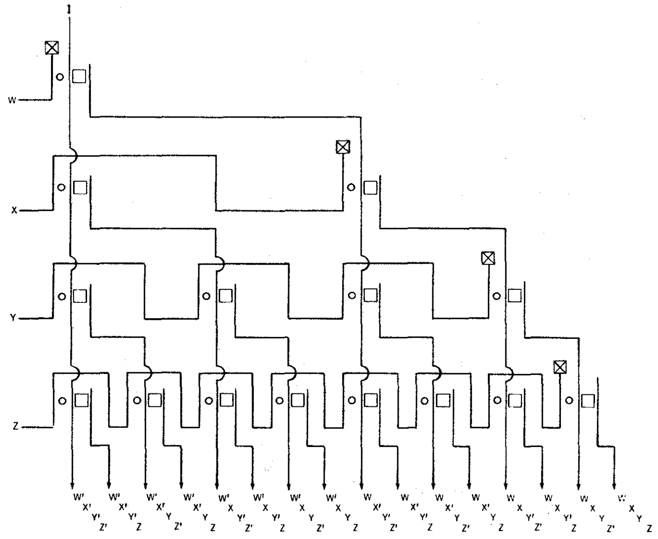


Figure 10—Decoder using class 21

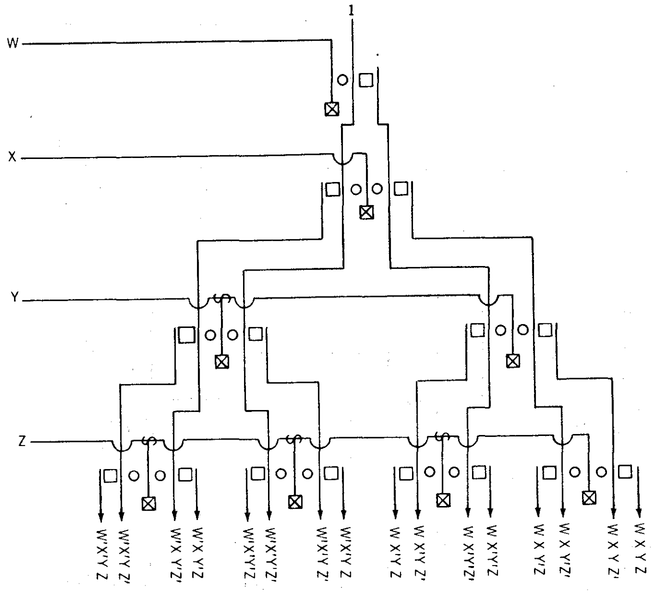


Figure 11—Alternative decoder using class 21

Figure 12(A) illustrates a two-variable decoder with no crossovers. In order to assemble decoders which handle high numbers of variables from this two-variable decoder, the planar crosspoint circuit shown in Figure 12(B) is useful. By putting the circuits of Figure 12(A) and (B) together as shown in Figure 12(C), a four-variable decoder results. It should be emphasized that this decoder has no physical bubble-track crossovers. If it is desired to matrix the 16 outputs of two decoders similar to the one in Figure 12(C) in order to form an eight-variable decoder and still to retain planarity, it will be necessary to use logical crossover circuits similar to the one shown in Figure 7(A) in order to bring the 16 outputs of Figure 12(C) into a row so that they can be matrixed.

MAGNETIC BUBBLE COMPUTER

Assumptions

It is the belief of the authors that a sufficient variety of bubble logic has been described so that any digital system can be efficiently designed. To illustrate this conviction, a sketch for the design of a magnetic bubble computer will be given. Such a design exercise will additionally show the structure of various bubble digital subsystems and expose those areas in which further work needs to be done.

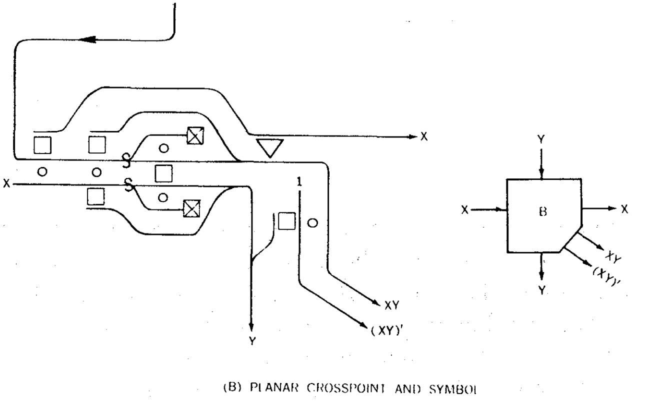
Since the speed of propagation in magnetic bubble circuits is presently in the range of 5 to 300 meters/sec., which is several orders of magnitude slower than the propagation speed for electrical signals in wire, one design criterion is that the over-all operating speeds of the bubble computer be realistic. For similar reasons, the memory size must be realistic. On the other hand, no useful purpose is served in this design exercise by including an input-output system or an interrupt structure.

Specifications

The memory for this computer consists of 16 serial shift registers which are designed to incorporate a small amount of logic. These shift registers are called *mark-time lines*. Each shift register holds 256 words of 16 bits each, and the address capability is provided to lengthen each shift register to 65,536 words. Therefore, the memory size ranges from the 4,096 words shown in the figures that follow, to 1,048,576 words.

The number system is *two's complement* with the sign bit inverted. This number system is particularly suitable for asynchronous operations because all valid words have at least one binary 1.

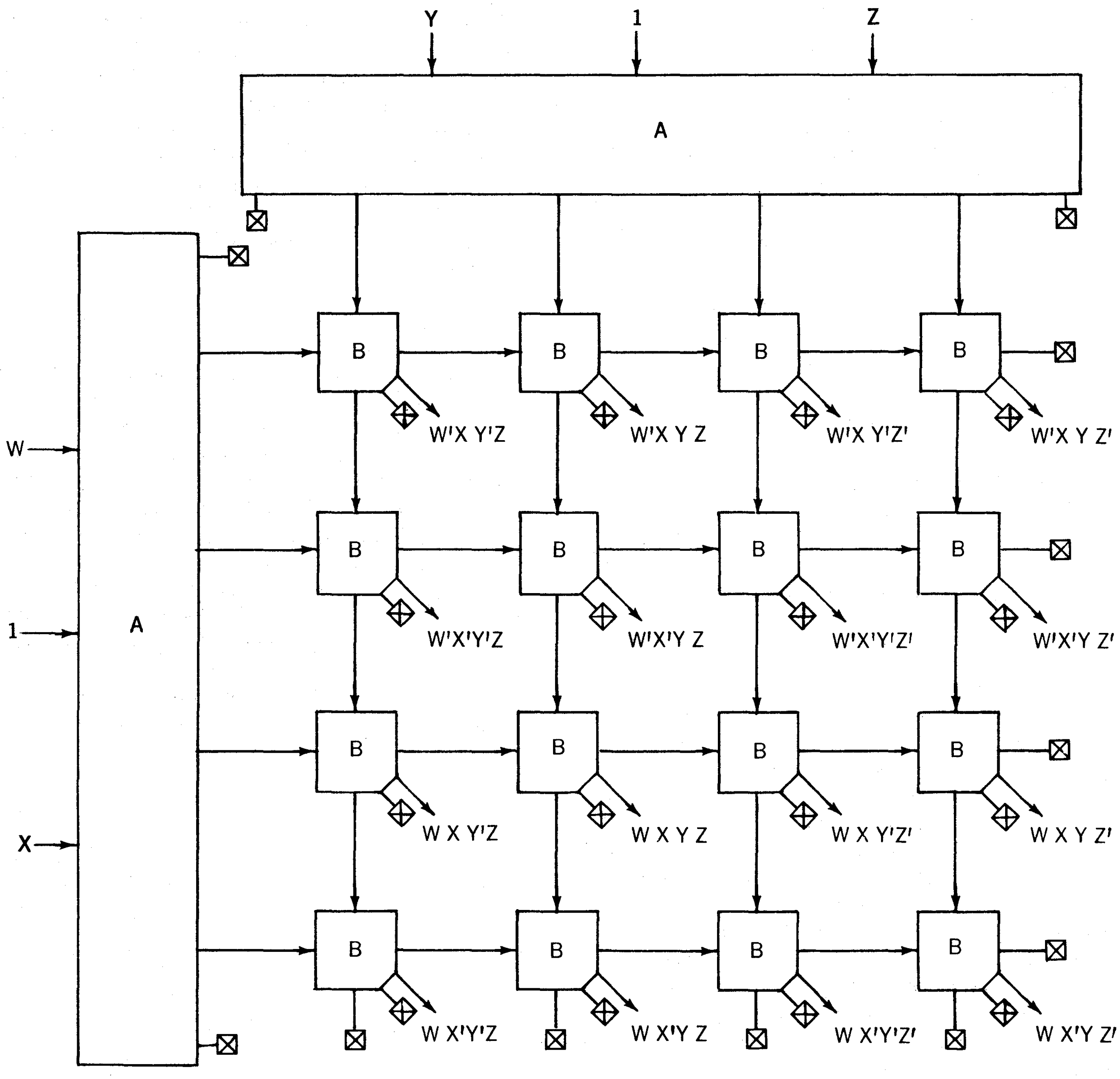
There are 32 logical sub-systems in the bubble computer: they are called *modules*. Sixteen of these



(A) PLANAR TWO-VARIABLE DECODER AND SYMBOL

(B) PLANAR CROSSPOINT AND SYMBOL

Figure 12—Planar decoders



(C) PLANAR FOUR-VARIABLE DECODER

Figure 12—(continued)

modules constitute logical interfaces between the 16 shift registers and the arithmetic portions of the computer. The other 16 modules are specialized to various arithmetic, logical and control functions. Some of these contain one or several storage registers as appropriate. The modules are interconnected by a closed buss called a *daisy chain.*

The instruction format is shown as Figure 13. All

DC	DA	SC	SA

SA: Address of Source Module
SC: Source Module Control Field
DA: Address of Destination Module
DC: Destination Module Control Field

Figure 13—Instruction format

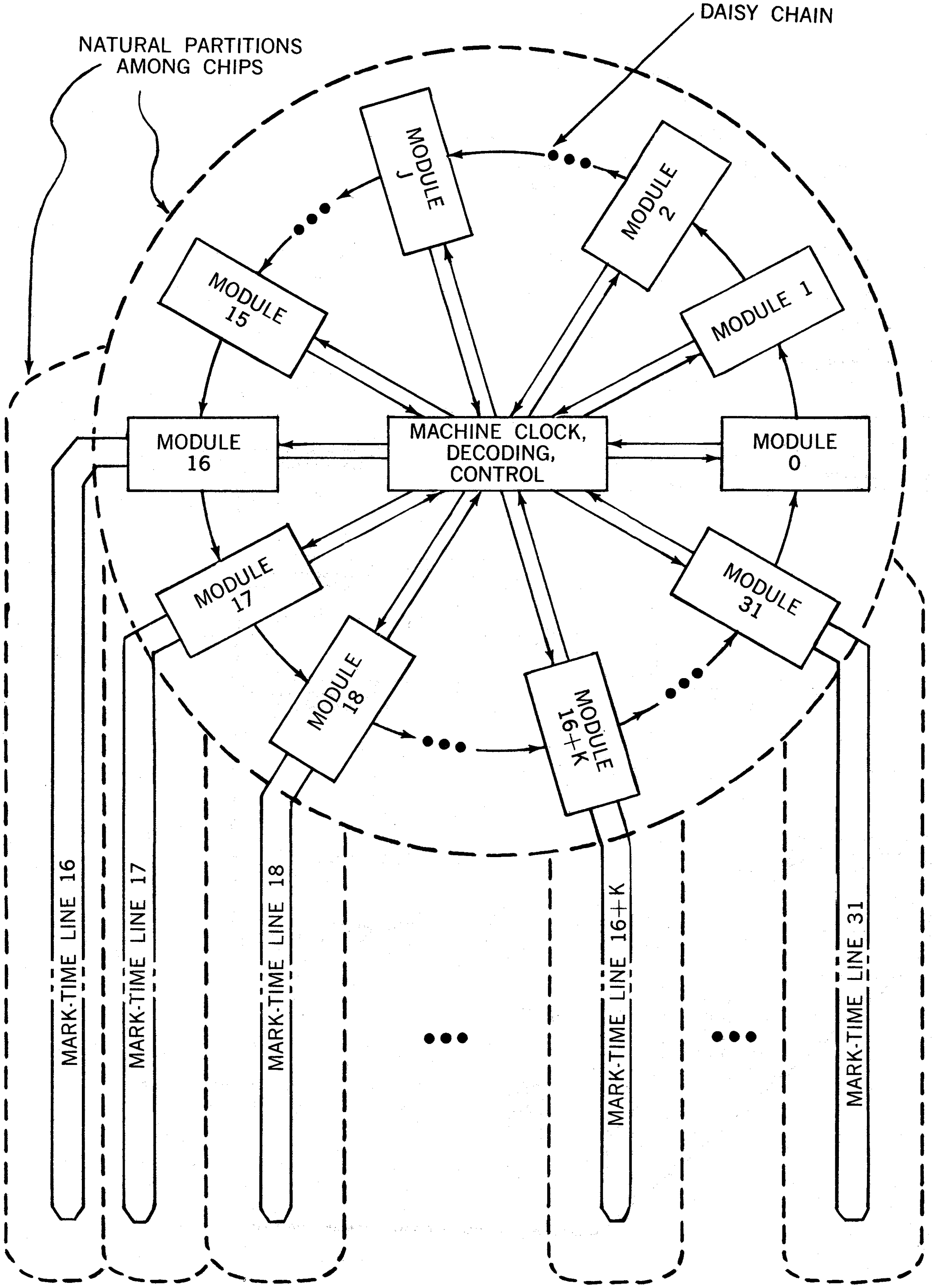


Figure 14—Bubble computer block diagram

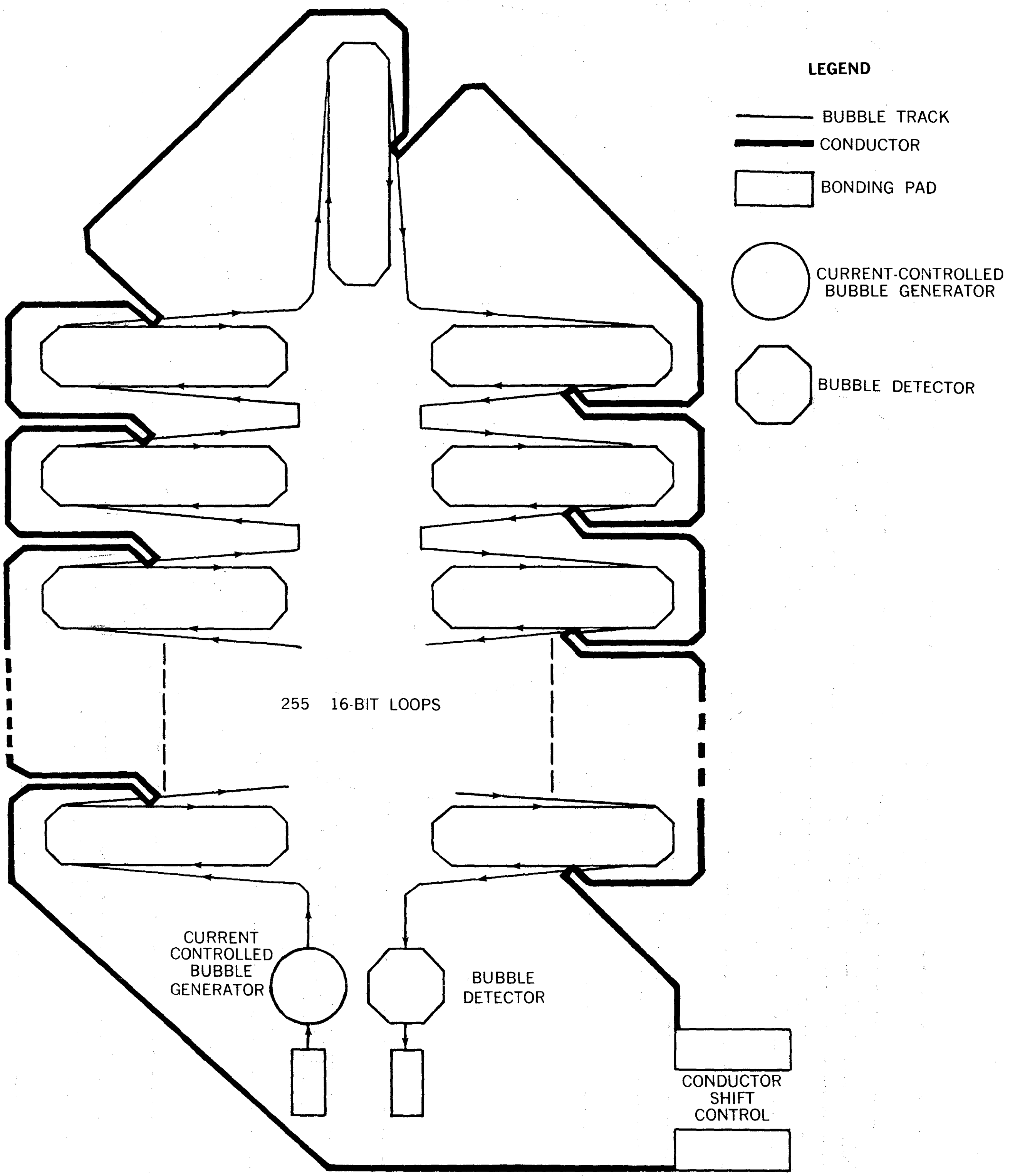


Figure 15—Schematic of mark-time line

instructions consist of designating a transfer from one of the 32 modules to another module, or of initiating a search for a word location in a mark-time line. Furthermore, each of the source and destination modules receives a three-bit control which is used to designate the specific module actions during the transfer.

The computer control utilizes two of the modules for designating the line from which instructions presently are being delivered, and for buffering the instruction which is being executed.

A block diagram for this bubble computer is given as Figure 14.

Mark-time lines

Magnetic bubble shift registers have been extensively studied, and their design is well-known. Therefore, a modified shift register storage has been chosen; it is shown schematically in Figure 15. There are both bubble tracks and current conductors in this figure; in order to avoid confusion, bubble tracks appear as narrow lines.

In the absence of a current applied to the conductor shift control in Figure 15, the bubbles in the memory move around the 255 closed 16-bit loops. At the end of one word time, the whole memory is in the same condition it was in the previous word time. Thus, even though the drive is continually applied, the words are effectively marking time: therefore, the name. On the other hand, if a current is applied to the conductor shift control in Figure 15, information advances from the output paths of each 16-bit loop into the next loop. Therefore, the conductor shift control current determines whether the words advance around the delay line or mark time. The principle of the weak S-curve[1] is used in the mark-time line.

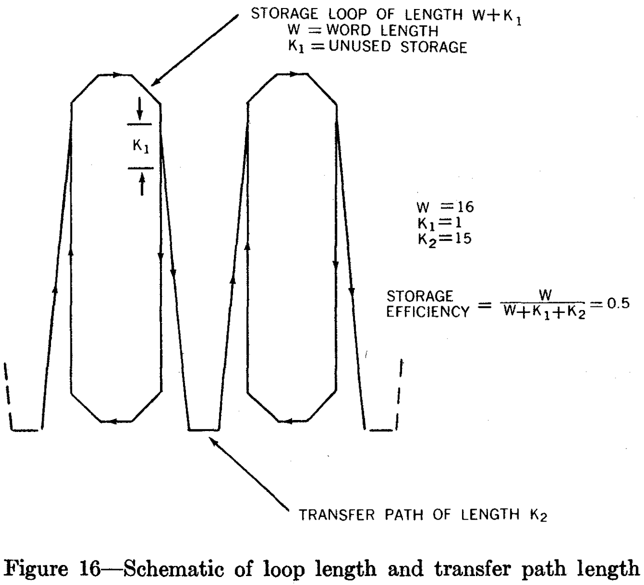

STORAGE LOOP OF LENGTH W+K_1
W = WORD LENGTH
K_1 = UNUSED STORAGE
K_1
W =16
K_1=1
K_2=15
STORAGE EFFICIENCY = $\frac{W}{W+K_1+K_2}$ = 0.5
TRANSFER PATH OF LENGTH K_2

Figure 16—Schematic of loop length and transfer path length

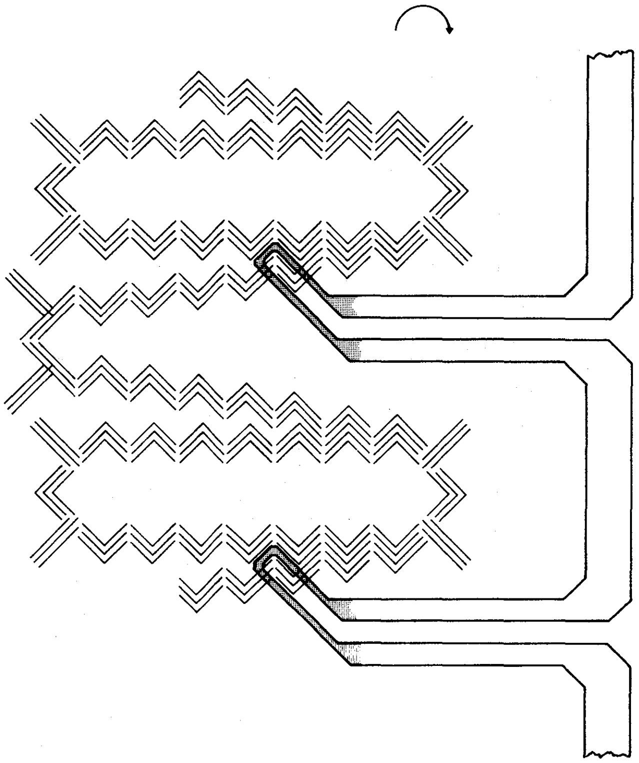


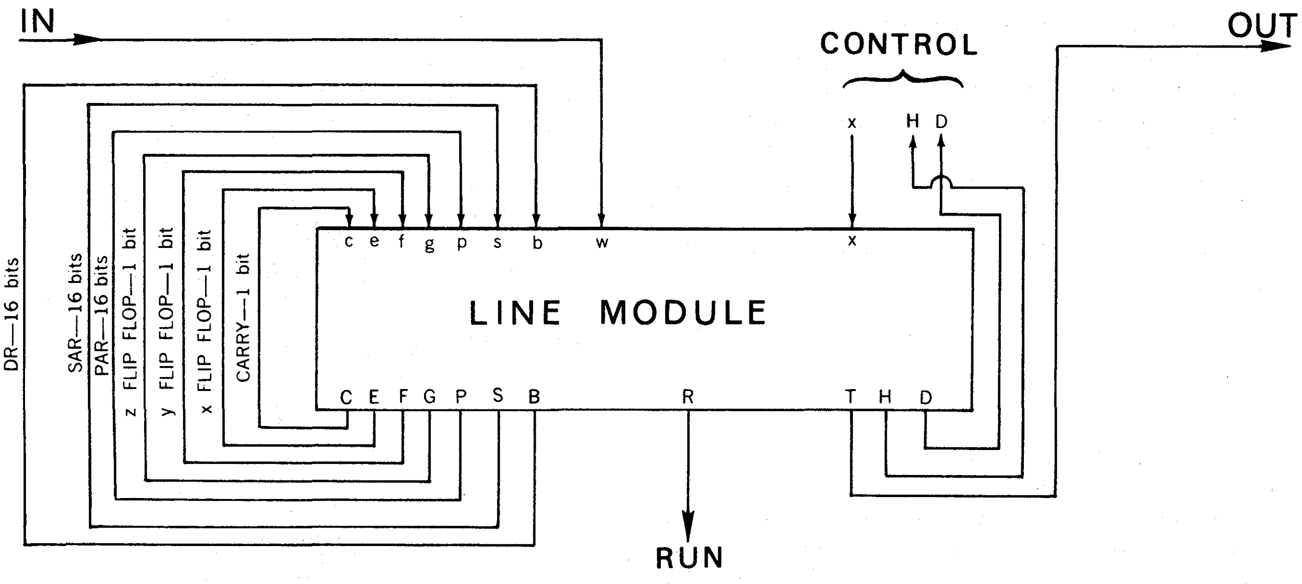
Figure 17—Realization for the mark-time line

Figure 16 shows some of the design parameters in this storage device, and Figure 17 is a multi-chevron layout for two loops.

Line modules

It is seen from Figure 14 that each of the 16 mark-time lines is connected to one of modules 16 through 31. These memory-interface modules are termed *line modules*.

Each line module contains one word of the mark-time line: it is called the *datum register*, or DR. In addition, there are two serial one-word registers associated with each line module, the *present address register* (PAR) and



(A) BLOCK DIAGRAM

x	y	z	OUTPUT	PAR	SAR	DR
0	0	0	IN → OUT	Recirculate	Recirculate	Recirculate
0	0	1	0 → OUT	Recirculate	Write	Recirculate
0	1	0	DR → OUT	Recirculate	Match PAR	Read
0	1	1	SAR → OUT	Recirculate	Read	Recirculate
1	0	0	0 → OUT	Write	Recirculate	Recirculate
1	0	1	0 → OUT	Recirculate	Recirculate	Write
1	1	0	PAR → OUT	Read	Recirculate	Recirculate
1	1	1	DR → OUT	Count	Count	Recirculate

(B) CONTROL CODES

$E = xP_{29} + e(cP_{28})'$
$F = xP_{30} + f(cP_{28})'$
$G = xP_{31} + g(cP_{28})'$
$T = e' f' g' w + e' f g s + e f g' p + f (e \oplus g') b$
$P = p [e' + (f \oplus g)] + f' g' w + e f g [p \oplus c \oplus (x z)]$
$S = s [f' g' + (e \oplus f \oplus g)'] + e' f' w + e f g [p \oplus c \oplus (x z)]$
$C = [e f g \text{ Maj } (p, c, x z) + f' (e \oplus g) (w + c) + e f' g c (p \oplus s)] P_{28}'$
$R = f (e \oplus g')$
$D = [f + c (e + g)] P_{28}$
$H = e + f + g$
$B = (e' + f + g') b + e f' g w$

(C) LOGICAL EQUATIONS

Figure 18—Line module

the *search address register* (SAR). The present address register contains the address of the word currently in the datum register. Because each mark-time line is started and stopped under control of its line module, the ncrementing of the present address register is similarly controlled in order to keep track of the present memory address.

It is well-known that even though random-access memories are in common use today, memory accesses are frequently too sequentially organized items. That is, the program often is stored in segments, each of which is in contiguous memory locations. Similarly, related data such as matrices often are stored adjacently in memory. For these reasons, it is believed that the mark-time lines will be accessed for the next word in sequence much of the time, and only occasionally for a word at a random location.

In order to provide for the occasional random access to a word in a line, the address of the desired word is loaded into the search address register of its line module, and it in turn provides the conductor shift control called the *run signal* to the mark-time line. When the present address register equals the search address register, the desired word is located in the datum register of the line module and the run signal is automatically removed.

Because this searching process for the head of a new set of contiguous data or program segment may take a significant amount of time, the line modules are organized such that this operation may proceed autonomously of other operations on the daisy chain. Therefore, the programmer may arrange to have such sets of data or program in separate mark-time lines, and start the search procedure for the head of any such set sufficiently prior to its need.

Some of the details of the line module are shown as Figure 18. In addition to the three one-word registers DR, SAR, and PAR, there are three flip-flops which buffer the three-bit sequential control field x. Depending on the control code, the module is arranged to perform one of eight actions as detailed in Figure 18(B). It should be noted that the absence of a control code corresponds to $x = (0, 0, 0)$; in this case, the input is transferred to the output and the various registers are recirculated. The logical equations appear as Figure 18(C).

All flip-flops are reset at bit time P_{22}. The control system initiates the action in a line module by delivering the three sequential control bits x at times P_{22}, P_{30}, and P_{31}, as shown in Figure 18. If a given line module is a receiving module, then when it has received any word destined for it on the daisy chain, a signal D is sent back to the control system. Therefore, the D signal indicates that the daisy chain is clear, and the control system may proceed with the next operation. However, if a search of a mark-time line is under way, an H signal is made true so as to delay any further control signals destined for the given module.

Special modules

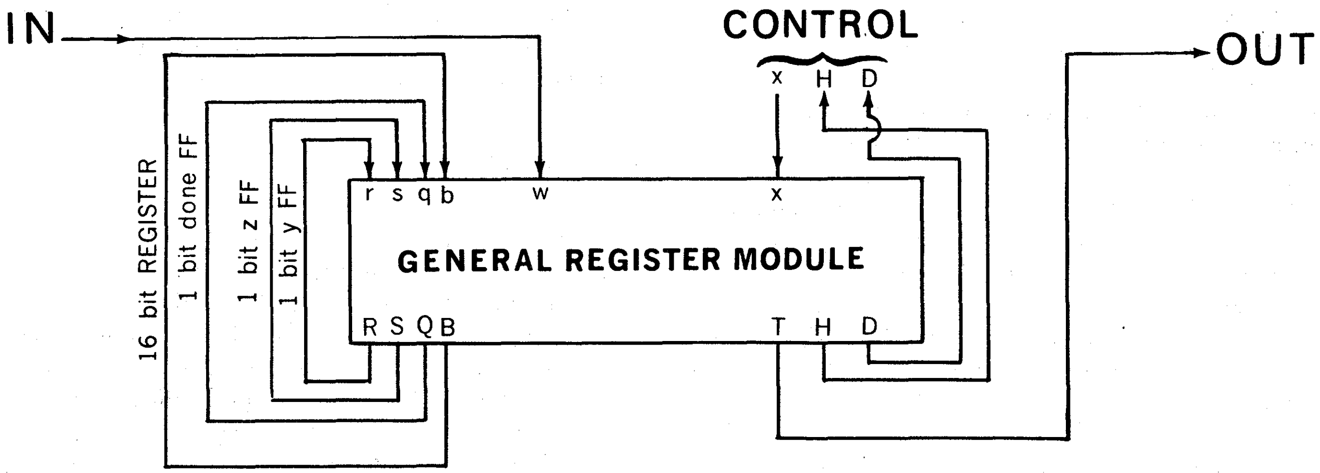
The remaining 16 modules on the daisy chain in Figure 14 serve various functions as shown in Figure 19. Module 0 holds a pointer to the mark-time line from which instructions presently are being obtained, while the fetched instruction is buffered in module 4. Modules 1, 2, and 3 are index registers which index instructions as they pass through on the daisy chain. Modules 5 through 8 are four identical arithmetic registers: each performs additions and subtractions. Modules 9 and 10 are used together for the purpose of shifting numbers that pass through. It is necessary to use two modules for shifting because of the limited number of control bits, and modules 12 through 15 are simple buffer registers which store intermediate results.

The block diagram for a general register module is shown as Figure 20(A). Only two of the three control bits are needed for this module, as shown in Figure 20(B), because four operations are sufficient for the desired actions. The logical equations for this module are given in Figure 20(C), and they are shown as bubble logic realizations in Figure 21.

Analogous details for the accumulator modules appear as Figure 22.

Module	Abbreviation	Name
M0	LNR	Line Number Register
M1	BR1	Index Register Number 1
M2	BR2	Index Register Number 2
M3	BR3	Index Register Number 3
M4	IR	Instruction Register
M5	AR1	Accumulator Register Number 1
M6	AR2	Accumulator Register Number 2
M7	AR3	Accumulator Register Number 3
M8	AR4	Accumulator Register Number 4
M9	SR1	Shift Register Number 1
M10	SR2	Shift Register Number 2
M11	MR	Immediate Register
M12	GR1	General Register Number 1
M13	GR2	General Register Number 2
M14	GR3	General Register Number 3
M15	GR4	General Register Number 4

Figure 19—Special modules



(A) BLOCK DIAGRAM

y	z	OUTPUT	REGISTER
0	0	IN ⟶ OUT	Recirculate
0	1	IN ⟶ OUT	Erase
1	0	0 ⟶ OUT	Write
1	1	REGISTER ⟶ OUT	Recirculate (Read)

(B) CONTROL CODES

$$\begin{aligned}
R &= xP_{30} + r\cdot(qP_{28})' \\
S &= xP_{31} + s\cdot(qP_{28})' \\
Q &= (w + q)\ P_{28}' \\
B &= (r \oplus s')\ b + r\ s'\ w \\
T &= r'\ w + r\ s\ b \\
D &= (qr + r's)\ P_{28} \\
H &= 0
\end{aligned}$$

(C) LOGICAL EQUATIONS

Figure 20—General Register module

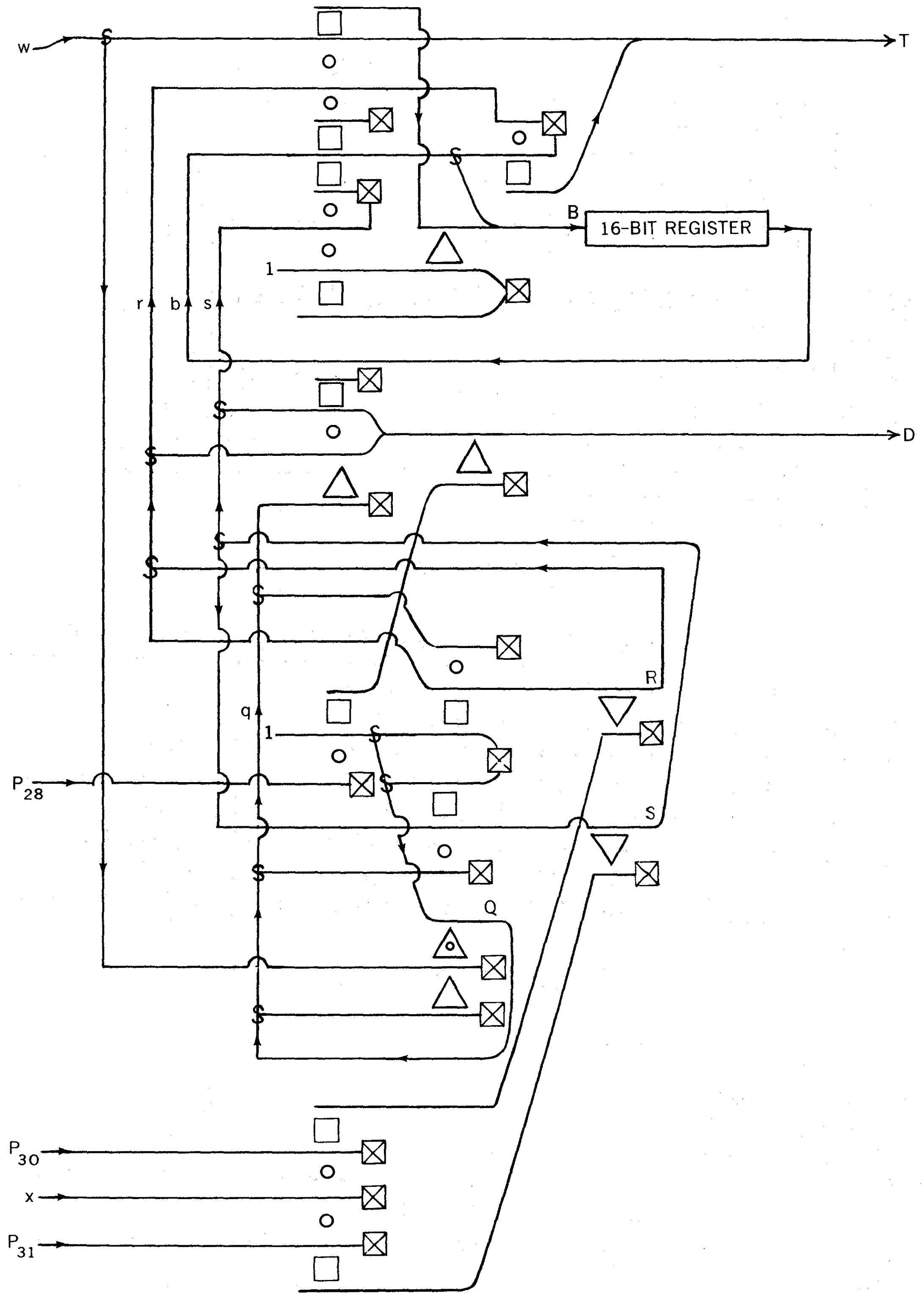
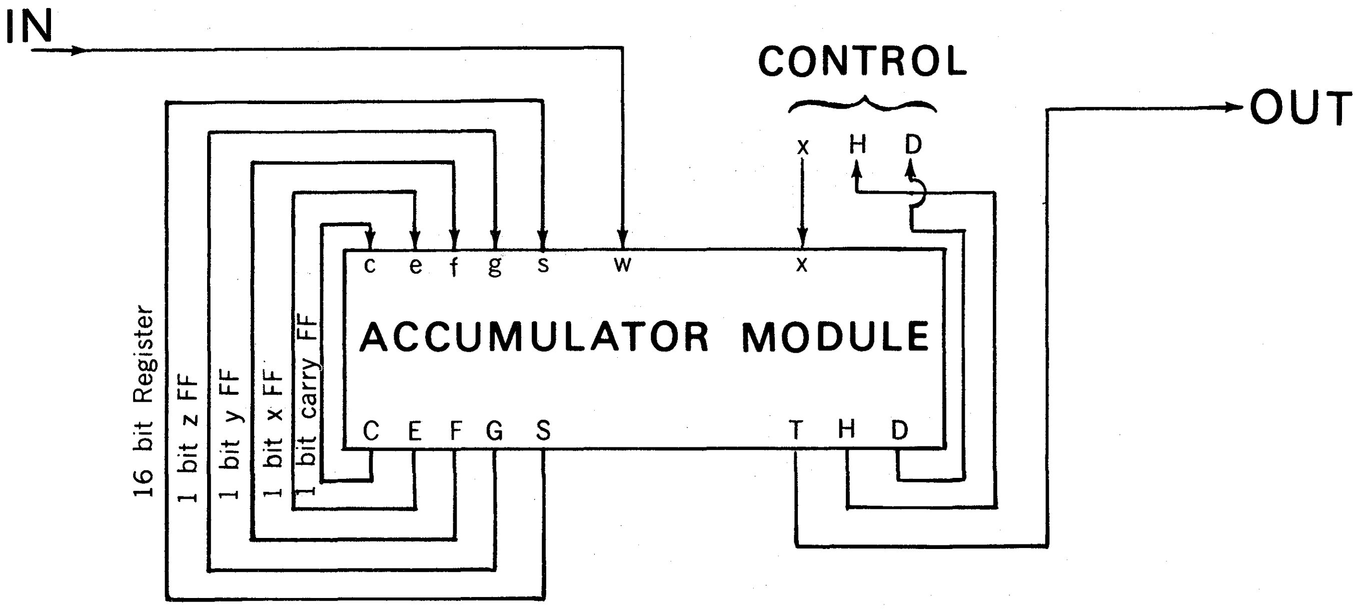


Figure 21—Bubble logic realization for the general register module



(A) BLOCK DIAGRAM

x	y	z	OUTPUT	REGISTER	OPERATION
0	0	0	IN ⟶ OUT	Recirculate	IDLE
0	0	1	0 ⟶ OUT	Erase	ERASE
0	1	0	0 ⟶ OUT	Write Input	WRITE
0	1	1	0 ⟶ OUT	Negative Write	NEG. WRITE
1	0	0	S ⟶ OUT	Recirculate	READ
1	0	1	-S ⟶ OUT	Recirculate	NEG. READ
1	1	0	0 ⟶ OUT	Add Input	ADD
1	1	1	0 ⟶ OUT	Subtract Input	SUBTRACT

(B) CONTROL CODES

$$
\begin{aligned}
E &= xP_{29} + e(cP_{28})' \\
F &= xP_{30} + f(cP_{28})' \\
G &= xP_{31} + g(cP_{28})' \\
T &= e'\,f'\,g'\,w + ef'\,g'\,s + ef'\,g\,(s' \oplus x) \\
S &= f'\,s\,(e + g') + e'\,fg'\,w + e'\,fg\,(w' \oplus y) + efg'\,(s \oplus w \oplus c \oplus P_{28}) \\
&\qquad + efg\,(s \oplus w' \oplus c \oplus z \oplus P_{28}) \\
C &= [\,efg'\,\mathrm{Maj}(s,\ w,\ c) + efg\,\mathrm{Maj}\,(s,\ w',\ c,\ z) + f'\,(e + g') + e'\,f\,(w + c)]\ P'_{28} \\
D &= P_{28}c \\
H &= 0
\end{aligned}
$$

(C) LOGICAL EQUATIONS

Figure 22—Accumulator module

Control circuits

The magnetic bubble computer operates on a conventional fetch-execute cycle. During the fetch phase the address of the line module from which the instruction is to be fetched is obtained from module 0 and decoded. Module 0 always designates module 4 as the destination for the instruction fetch; therefore, two decoding circuits serve to steer the source and destination control fields to the appropriate modules. Once module 4 has received its new instruction, it sends a D signal back to the control, indicating that the daisy chain buss is clear. This D signal switches the control system into the execute phase.

In the execute phase, the source and destination address fields from the instruction control the decoder to steer the source and destination control bits to the correct modules. When the destination module signals that the daisy chain buss is clear by sending a D signal back to the control circuits, the fetch phase is reentered.

The circuits associated with the control are shown in Figure 23. In Figure 23(A) is a trigger flip-flop similar to the one in Figure 9(A) which alternately steers the control of module 0 (the x signal) and of module 4 (the y signal) to the output X in accordance with the returned buss-clear control signal d (D at its sources). This accomplishes the fetch-execute control. In Figure 23(B) is a block diagram for a five-stage serial parallel converter, which converts the x signal (X of Figure 23(A)) into a five-bit parallel word. A bubble design for one stage of this converter is shown in Figure 23(C).

Each bubble output from the serial-parallel converter is converted into a sequence of three bubbles using the bubble tripler of Figure 23(D). This tripler is derived from the circuit of Figure 8(B); it causes the decoder to be held at its value for three bit times so that the control field can be steered to the correct module. Finally, the outputs of the five bubble triplers feed the T_i^3 inputs of the 32-output decoder in Figure 23(E). The C input of that figure is the three-bit control field.

Bubble chip area and speed considerations

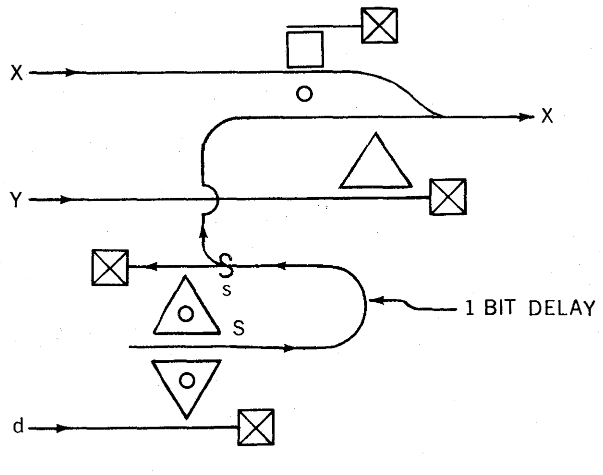
It is estimated that this computer would fit onto between six and 20 bubble chips. For the portions of the computer exclusive of the mark-time lines, the assumption is made that a logical circuit is approximately four circuit periods long and three high. Using several designs as samples, it appears that when all auxilary items such as generators, annihilators, bubble tracks, etc., are taken into account, the average area occupied by one gate is between 50 and 150 periods.^{2} Conservatively assuming 50 gates per module with 100 periods^2 per gate, approximately 100,000 periods^2 are needed for all the modules. An additional 20,000 periods^2 for the control circuits would be generous. As a result, 180,000 periods^2 are needed for the CPU; at the present-day capabilities of bubble chip sizes and circuits this would require two to four chips.

The area taken by one 256-word mark-time line is approximately 23,000 periods,^{2} therefore, one to four storage lines can be constructed on one chip using the technology available today.

In order to estimate the time for a complete fetch-execute cycle, it is assumed that each module presents a delay of 32 periods and that an additional 32 periods are needed for the control circuits. If reasonably good programming practice is used, then either the program or the data are stored in lines fairly near modules 0 and 4. The assumption is that the instructions are stored four modules away and the data 16 modules away. From these assumptions it follows that a fetch-execute cycle takes slightly less than 900 periods. At a 10^6-bit per second rate, this corresponds to 10^3 instructions executed per second.

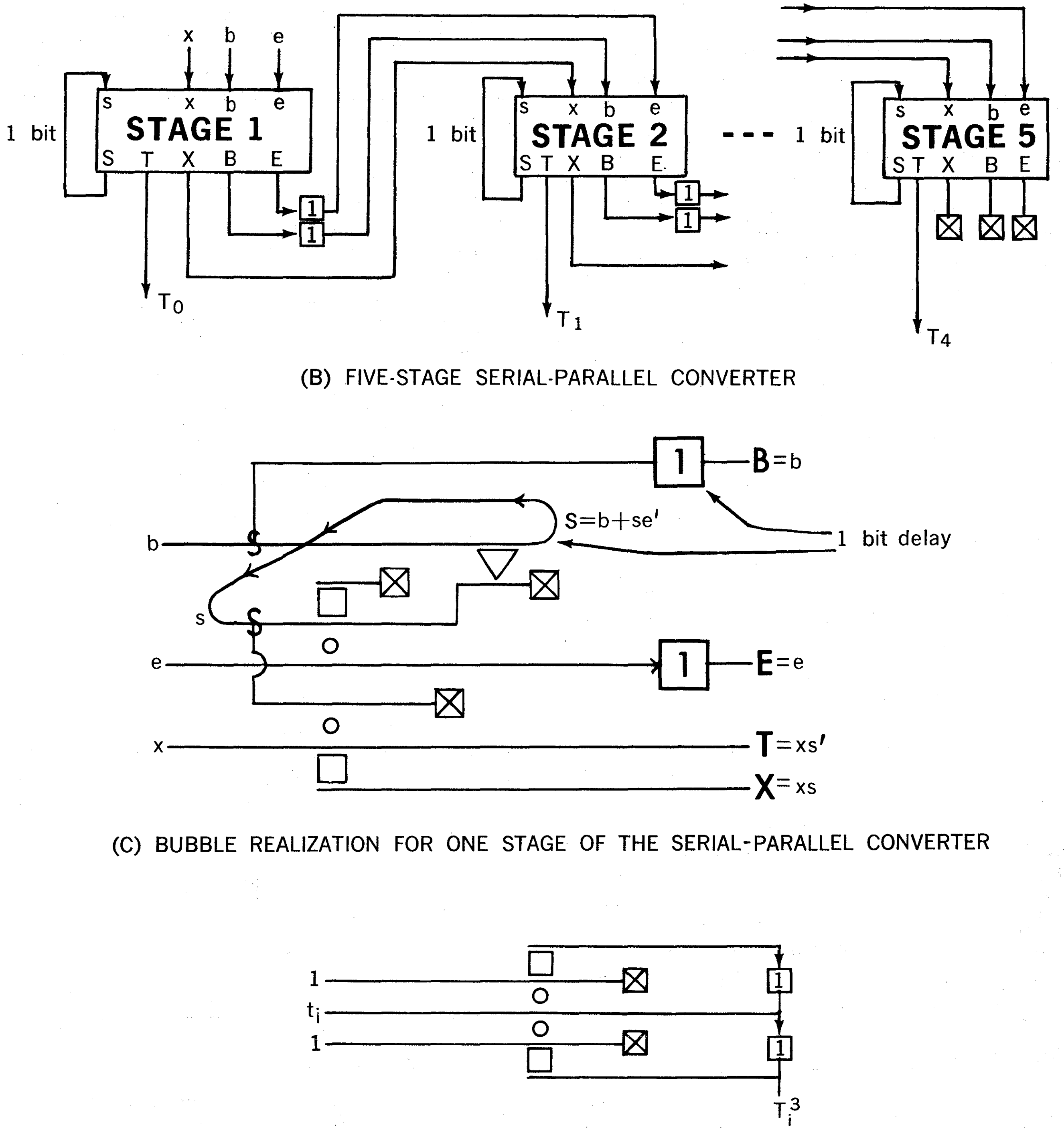
SUMMARY AND CONCLUSIONS

Several new bubble logic and memory circuits have been shown. As an exercise a magnetic bubble computer has been designed which should fit onto between six and 20 bubble chips. That such a design can be made demon-



(A) BUBBLE REALIZATION FOR FETCH-EXECUTE CONTROL

Figure 23—Control circuits

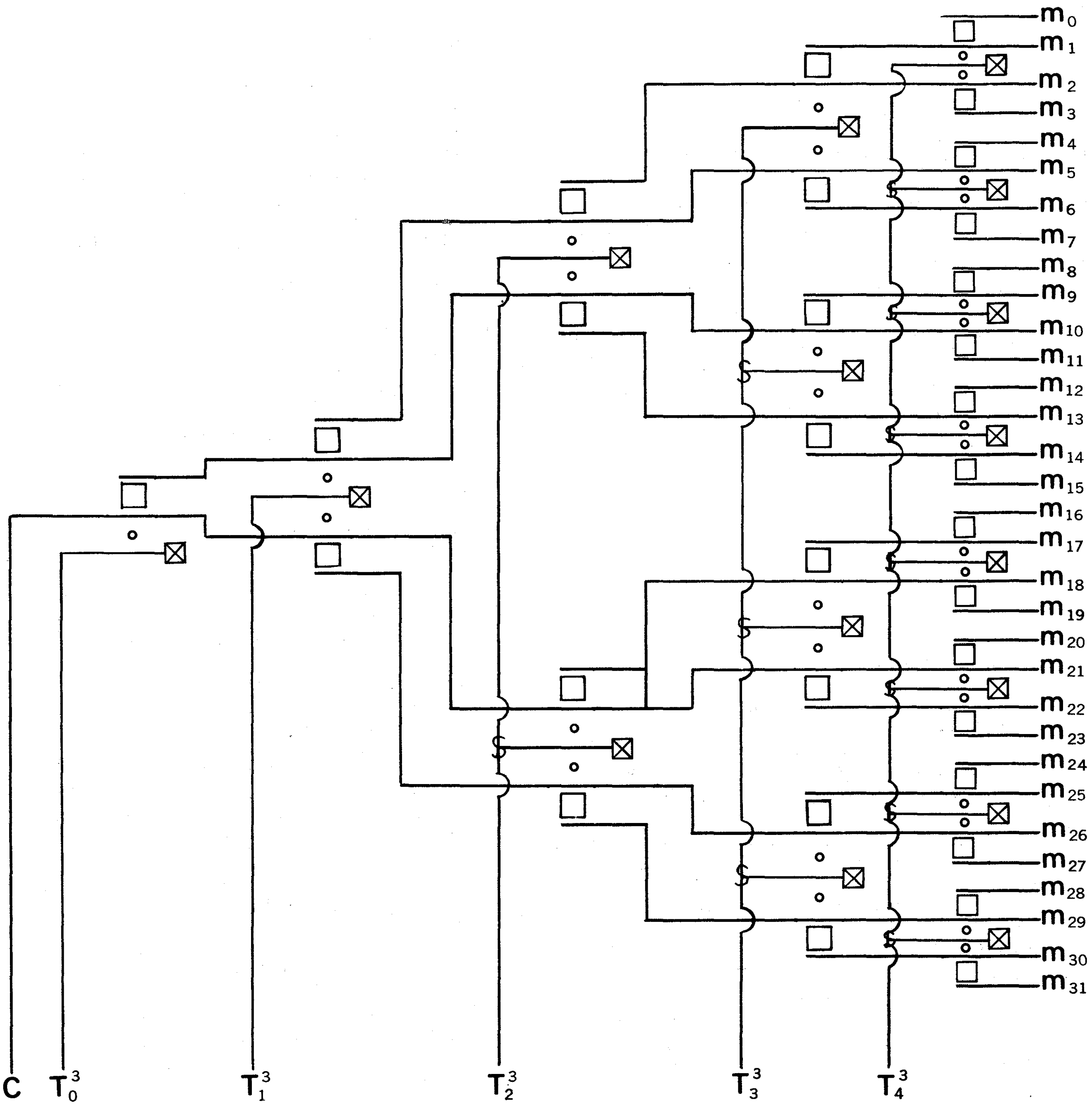


(B) FIVE-STAGE SERIAL-PARALLEL CONVERTER

(C) BUBBLE REALIZATION FOR ONE STAGE OF THE SERIAL-PARALLEL CONVERTER

(D) BUBBLE TRIPLER

Figure 23—(continued)



(E) THIRTY-TWO OUTPUT DECODER

Figure 23—(continued)

strates the completeness of the bubble logic that has been shown.

ACKNOWLEDGMENTS

The authors wish to acknowledge the continuing support and encouragement provided by Dr. Remo A. Pellin and Mr. George M. MacLeod, the laboratory work of Mr. Jerry W. Moody and Dr. Roger W. Shaw, the typing of Ms. Sandra S. Johnson, and the drafting of Mr. Ray Nehman and Mr. William B. Bierman.

Without the advice, suggestions, encouragement, and most importantly the drive supplied by Dr. Howard H. Aiken, this work would never have been begun or pursued or finished.

REFERENCES

1 R C MINNICK P T BAILEY R M SANDFORT W L SEMON
Magnetic bubble logic
WESCON Proceedings 1972

2 G S ALMASI et al
Fabrication and operation of a self-contained bubble-domain memory chip
American Institute of Physics Proceedings No 5 1972

3 A H BOBECK
Properties and device applications of magnetic domains in in orthoferrites
Bell System Technical Journal Vol 46 1967

4 A H BOBECK R F FISHER A J PERNESKI
A new approach to memory and logic-cylindrical domain devices
Proceedings of the Fall Joint Computer Conference 1969

5 A H BOBECK et al
Application of orthoferrites to domain-wall devices
IEEE Transactions on Magnetics Vol MAG-5 1969

6 A H BOBECK H E D SCOVIL
Magnetic bubbles
Scientific American June 1971

7 A H BOBECK R F FISHER J L SMITH
An overview of magnetic bubble domains—Material device interface
American Institute of Physics Conference Proceedings No. 5 1972

8 E A GIESS et al
Rare-earth-yttrium iron-gallium garnet epitaxial films for magnetic-bubble domain applications
Materials Research Bulletin Vol 6 1971

9 A A THIELE
A theory of cylindrical magnetic domains
Bell System Technical Journal Vol 48 1969

10 A H BOBECK
Recent developments in magnetic bubble technology
Electrochemical Society Meeting Houston 1972

11 P I BONYHARD et al
Applications of bubble devices
IEEE Transactions on Magnetics Vol MAG-6 1970

12 H N CARLSON et al
Field access bubble to bubble logic operations
Intermag Conference Proceedings 1972

13 J A COPELAND
U.S. patents 3641518 and 3653010

14 A D FRIEDMAN P R MENON
Mathematical models of computation using magnetic bubble interactions
Bell System Technical Journal 1971

15 M R CAREY
Resident-bubble cellular logic using magnetic domains
IEEE Transactions on Computers 1972

16 R M GOLDSTEIN M SHOJI
Functional bubble domain circuits employing bubble-bubble iteraction
American Institute of Physics Proceedings No. 5 1972

17 R L GRAHAM
A mathematical study of a model of magnetic domain iteractions
Bell System Technical Journal 1970

18 R P KURSHAN
All terminal bubbles programs yield the elementary symmetric polynomials
Bell System Technical Journal 1970

19 A J PERNESKI
Propagation of cylindrical magnetic domains in orthoferrites
IEEE Transactions on Magnetics Vol MAG-5 1969

20 R M SANDFORT E R BURKE
Logic functions for magnetic bubble devices
IEEE Transactions on Magnetics 1971

21 M SHOJI
Magnetic bubble counting circuits
IEEE Transactions on Magnetics Vol MAG-8 1972

22 J H SPENCER
(Unpublished)

23 W H KAUTZ K N LEVITT A WAKSMAN
Cellular interconnection arrays
IEEE Transactions on Computers Vol C-17 1968

Numerical solution of ill-posed problems using interactive graphics

by J. M. VARAH

University of British Columbia
Vancouver, B. C., Canada

INTRODUCTION

The problems we consider can all be expressed as integral equations of the first kind: i.e., given $K(s, t)$, $g(s)$, solve

$$g(s) = \int_a^b K(s, t)f(t)\, dt. \qquad (1)$$

Such problems are ill-posed in general because small changes in $g(s)$ can cause large changes in the high-order "modes" of $f(t)$. (This is explained below.) Examples of such problems are:

(i) harmonic continuation in a circle [$a=0$, $b=2\pi$, $K(s, t)=$Poisson kernel]
(ii) inversion of Laplace transforms [$a=0$, $b=\infty$, $K(s, t)=\exp(-st)$]
(iii) the backwards heat equation [$a=-\infty$, $b=\infty$, $K(s, t)=\exp(-(s-t)^2/4\tau)/\sqrt{4\pi\tau}$]

For $K(s, t)$ a compact operator (e.g., a, b finite, $K(s, t)$ bounded), we can express the solution to (1) in terms of the eigenfunctions of $K(s, t)$ if K is symmetric, or more generally in terms of the *adjoint orthogonal system* of K: i.e. $\{\phi_i(s), \psi_i(s)\}$ such that

$$\int K(s, t)\psi_i(t)\, dt = \lambda_i\phi_i(s)$$

$$\int K(t, s)\phi_i(t)\, dt = \lambda_i\psi_i(s).$$

The eigenvalues $\lambda_i \rightarrow 0$, and if $\gamma_i = \int g(s)\phi_i(s)\, ds$, then the solution to (1) is

$$f(t) = \sum \frac{\gamma_i}{\lambda_i}\psi_i(t). \qquad (2)$$

Notice that if $g(s)$ is changed by a small function $\eta(s)$ with $\int[\eta(s)]^2\, ds = \epsilon^2$, each γ_i may be changed by as much as ϵ. But since $\lambda_i \rightarrow 0$, this means for i large enough the component of $\psi_i(t)$ in (2) (the high order modes of $f(t)$) can become arbitrarily large.

It is well-known that a solution to (1) exists in l_2 if and only if $\Sigma\gamma_i^2/\lambda_i^2 < \infty$ (see e.g., Courant and Hilbert,[1] pg. 159ff for details). It is clear that some kind of restriction like this on $g(s)$ is necessary to have a well-posed problem. In the next section we will see how to enforce this restriction numerically.

NUMERICAL APPROXIMATION

Similar work has been done on this problem by Baker et. al.[2] and Hanson.[3] Using the quadrature rule

$$\int_a^b p(x)\, dx \cong \sum_1^n w_j p(x_j),$$

we approximate (1) at a discrete set of points $\{s_i\}_1^m$, giving a set of linear algebraic equations:

$$g(s_i) = \sum_{j=1}^n w_j K(s_i, t_j) f_j, \quad i=1, \ldots, m \qquad (3)$$

If $m=n$ and the matrix is nonsingular, this has a unique solution f whose components f_j are approximations to $f(x_j)$. We can then interpolate this data (for example with a cubic spline) to give an approximate solution throughout $[a, b]$.

However by its very nature the linear system $Af=g$ in (3) is ill-conditioned and if we solve by the usual Gaussian elimination routine, we will have

$$\text{error} \sim \eta_1/\sigma_n + (\text{truncation error}).$$

Here η_1 is the machine roundoff level and $1=\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n > 0$ are the singular values of A, with σ_n very small, especially if n is large. It seems clear that Gaussian elimination is not the natural way to solve this problem computationally, since it can be viewed as finding the exact solution to a slightly different problem, where the perturbation is essentially random. As we say earlier, such a small change in the data can cause a large, unwanted change in the high-order modes of the solution.

A much more natural method of solution is via the singular value decomposition; this factors the matrix into $A=UDV^T$, where U and V are orthogonal and $D=\text{diag}(\sigma_i)$. Moreover this can be done in a reasonable amount of time, roughly four times that of a Gaussian elimination routine, using the algorithm of Golub.[4]

Thus $Af=g$ becomes the uncoupled equations

$$D(V^Tf)=(U^Tg)$$

or, if $g=\sum_1^m \beta_i u_i$, (u_i = columns of U, v_i = columns of V)

$$f=\sum_1^n (\beta_i/\sigma_i)v_i \tag{4}$$

Now because of the relationships $K{:}A$, $\phi_i{:}u_i$, $\psi_i{:}v_i$, $\gamma_i{:}\beta_i$, $\lambda_i{:}\sigma_i$, restricting our problem to make it well-posed (in the sense of the last section) is equivalent to requiring $\beta_i/\sigma_i \rightarrow 0$ as i increases. This means f is closely approximated by

$$f^{(k)}=\sum_1^k (\beta_i/\sigma_i)v_i.$$

In fact

$$\|f-f^{(k)}\|_2^2=\sum_{k+1}^n (\beta_i/\sigma_i)^2,$$

and moreover[5] the roundoff error in computing $f^{(k)}$ depends only on η_1/σ_k, not σ_n. Hence

total error$\sim$(truncation error)

$$+\eta_1/\sigma_k+\left(\sum_{k+1}^n (\beta_i/\sigma_i)^2\right)^{1/2}, \tag{5}$$

giving some optimal solution $f^{(k)}$, $1\leq k\leq n$.

INTERACTIVE SOLUTION

How do we find the optimal k? We can estimate it by minimizing over k a bound for the last two terms of (5), but a more fruitful approach is to plot the solutions for various values of k using an interactive graphics terminal. Then the user can choose *visually* among these solutions for the best one. This of course may involve some rather subjective decisions, but we assume the user has some a priori idea of what his solution looks like.

There are many other ways of varying parameters in this problem: the choice of points $\{s_i\}$ and the quadrature rule for example. Since it is very complex to decide analytically which choice is best, the user may wish to experiment with various schemes and data points to optimize his solution.

Such an interactive program has been written for the

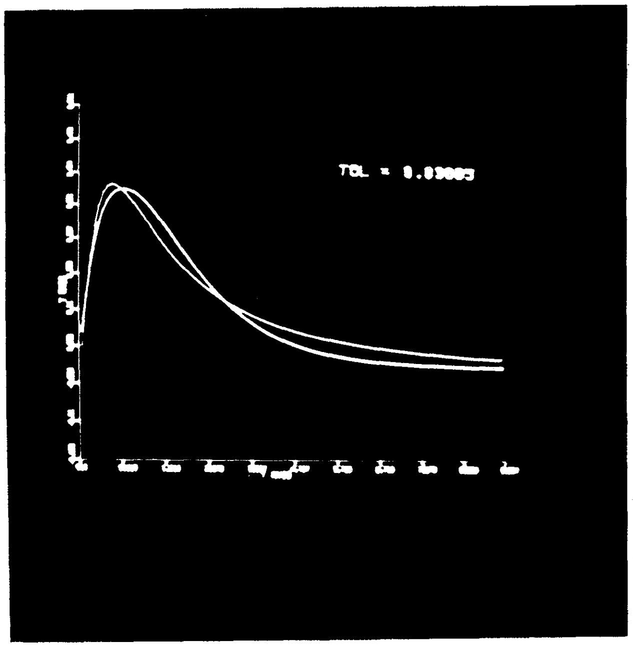


Figure 1

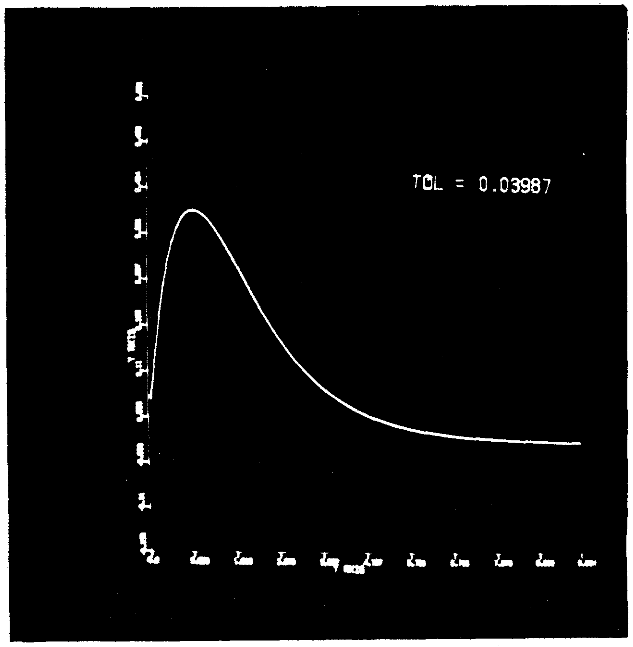


Figure 2

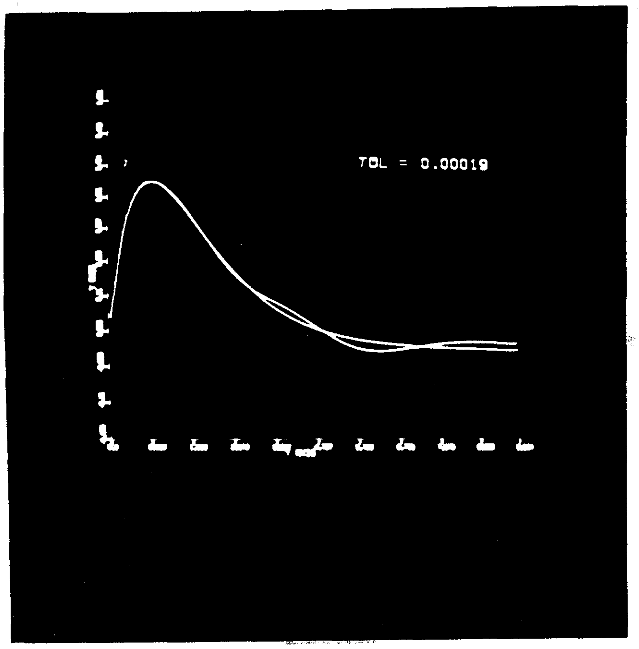


Figure 3

Adage model 10 graphics terminal coupled with the IBM 360/67 under MTS at U.B.C. We give one example here, that of Laplace transform inversion:

$$g(s) = \int_0^\infty e^{-st} f(t)\, dt.$$

The kernel here is *not* compact, so we have a continuous spectrum and the analysis of the first section does not hold. However, the solution $f(t)$ can still be well approximated by a linear combination (4) of approximations to the eigenfunctions corresponding to the low-order part of the spectrum.

For the particular case of $g(s) = 1/(s+1)^2$, with known solution $f(t) = te^{-t}$, and using Gauss-Lagueurre 20-point quadrature in (3), best results were obtained with the $\{s_i\}$ equally spaced in $(0, 5)$. Choosing $f^{(6)}$ as our solution (i.e., using 6 equations) gave a maximum error $|f - f^{(6)}| = 5 \cdot 10^{-4}$. Displayed here are the graphs of $f^{(3)}$, $f^{(6)}$, and $f^{(9)}$ produced on the graphics terminal. Cubic spline interpolation is used to define the functions between the Gauss-Lagueurre abscissas. Other numerical examples are given in another paper.[5]

REFERENCES

1 COURANT HILBERT
Methods of mathematical physics Vol I
Interscience New York 1953

2 C T H BAKER L FOX D F MAYERS K WRIGHT
Numerical solution of Fredholm integral equations of first kind
Computer Journal Vol 7 No 1964

3 R J HANSON
A numerical method for solving Fredholm integral equations of first kind using singular values
Siam Journal of Numerical Analysis Vol 8 No 4 1971

4 G H GOLUB C REINSCH
Singular value decomposition and least squares solutions
Numerische Mathematik Vol 14 No 1970

5 J M VARAH
On the numerical solution of ill-conditioned linear systems with applications to ill-posed problems
Siam Journal of Numerical Analysis Vol 10 No 1 1973

Iterative solution of elliptic difference equations using fast direct methods*

by PAUL CONCUS

University of California
Berkeley, California

INTRODUCTION

In recent years, fast direct methods have been developed for the numerical solution of the Poisson equation on a rectangle.[1,2] By taking advantage of the special block structure of the approximating discrete equation on a uniform rectangular mesh, these methods obtain the solution with striking efficiency and accuracy. A comparison of fast direct methods with other methods can be found in Reference 3, and the extension to more general separable elliptic equations in Reference 4.

Here, a technique is discussed for using fast direct methods to solve iteratively certain more general formally self-adjoint strongly elliptic equations $\mathcal{L}u=f$, which are not necessarily separable. Dirichlet conditions on the boundary of the rectangle are considered, although the technique applies with slight alteration to other boundary conditions for which fast methods are suitable. The approach is to utilize a modified form of the iterative procedure

$$-\Delta u_{n+1}=-\Delta u_n-\tau(\mathcal{L}u_n-f), \quad \Delta\equiv\partial^2/\partial x^2+\partial^2/\partial y^2 \quad (1)$$

proposed for numerical computation in conjunction with alternating-direction methods by D'yakonov[5] and discussed recently by Widlund.[6] This procedure, in addition to being of a form suitable for fast direct methods, has the desirable feature that for well-behaved problems its convergence rate is essentially independent of mesh size.

As it stands, however, iteration (1) may be too slowly convergent to be of practical importance, even when optimal values of the parameter τ are used. The means employed in this paper for improving its convergence rate are: (i) scaling the original problem $\mathcal{L}u=f$ and iterating instead with the scaled problem $\mathfrak{M}w=q$; (ii) using, instead of (1), the shifted iteration

$$(-\Delta+K)w_{n+1}=(-\Delta+K)w_n-\tau(\mathfrak{M}w_n-q), \quad (2)$$

where K is a suitably chosen constant; (iii) applying Chebyshev acceleration. Algorithms for the fast direct solution of the discrete Poisson equation in a rectangle can handle iteration (2), which requires the repeated solution of a Helmholtz equation, with the same rapidity as they can (1).

Related iterative techniques for elliptic equations are studied in References 7 and 8 in connection with alternating-direction methods and in References 9 and 10 in connection with Stone's sparse factorization method. This latter method is formally similar to the one here; however, the present technique has the desirable property of being based on a more natural splitting of the operator.

ITERATIVE PROCEDURE

Description

In its simplest form, the iterative procedure considered here solves numerically on a uniform rectangular mesh the problem

$$\mathcal{L}u\equiv-\nabla\cdot[a(x,y)\nabla u]=f(x,y) \qquad \text{on } \mathcal{R} \quad (3)$$

$$u(x,y)=g(x,y) \qquad \text{on } \partial\mathcal{R}, \quad (4)$$

where $\mathcal{R}$ is the rectangle $0<x<c$, $0<y<d$ and $a(x,y)$ is strictly positive on $\mathcal{R}$ and its boundary $\partial\mathcal{R}$. [It is assumed that $a(x,y)$, $f(x,y)$, and $g(x,y)$ are such that the solution $u(x,y)$ is sufficiently well behaved near the corners of $\mathcal{R}$ so that special numerical methods are not required there.] The positivity of $a(x,y)$ implies that $\mathcal{L}$ is positive definite.

* This work was supported in part by the U.S. Atomic Energy Commission.

If $a(x, y)$ has bounded second derivatives on the closed rectangle, which is the case of principal interest for use of the procedure, the change of variable is performed.

$$w(x, y) = [a(x, y)]^{1/2} u(x, y). \qquad (5)$$

Then, after division by $a^{1/2}$, (3) becomes

$$a^{-1/2}\mathcal{L}u = \mathfrak{M}w \equiv -\Delta w + p(x, y) w = q(x, y) \quad \text{on } \mathcal{R}, \qquad (6)$$

where $p(x, y) = a^{-1/2}\Delta(a^{1/2})$ and $q(x, y) = a^{-1/2}f$. The effect of this scaling is to transform the operator $\mathcal{L}$ into one whose differential part is $-\Delta$. Note that the change of variable (5) does not alter the positive definiteness of $\mathcal{L}$, so that $\mathfrak{M}$ is positive definite as well.

Substitution of (6) into (2) then yields as the iteration

$$(-\Delta + K) w_{n+1} = (-\Delta + K) w_n - \tau(-\Delta + p) w_n + \tau q \text{ on } \mathcal{R}. \qquad (7)$$

The boundary condition is

$$w_{n+1} = H(x, y) \quad \text{on } \partial\mathcal{R}, \qquad (8)$$

where $H(x, y) = a^{1/2} g$.

In an attempt to make the operator $-\Delta + K$ on the left of (7) agree closely with $\mathfrak{M}$, the constant K is chosen to approximate $p(x, y)$. The choice of central interest in this study is the minimax value,

$$K = (\beta + B)/2, \qquad (9)$$

where β is the minimum and B the maximum value of $p(x, y)$ on the closed rectangle. As will be shown in the next section, this choice leads to an estimate that the optimal value of the single parameter τ to give most rapid convergence in (7) is

$$\tau = 1. \qquad (10)$$

For this value of τ, (7) becomes simply

$$(-\Delta + K) w_{n+1} = (K - p) w_n + q \quad \text{on } \mathcal{R}. \qquad (11)$$

The discrete form of the iterative procedure (8, 9, 11) is obtained by placing a uniform rectangular mesh on $\mathcal{R}$ with spacing h in the x-direction and k in the y-direction and letting W_{ij} correspond to $w(x, y)$ at the mesh points $x = ih$, $y = jk$. Using the standard five point approximation for the operator $-\Delta$ with Dirichlet boundary conditions

$$-\Delta_h W_{ij} \equiv h^{-2}(-W_{i-1,j} + 2W_{ij} - W_{i+1,j}) + k^{-2}(-W_{i,j-1} + 2W_{ij} - W_{i,j+1}), \qquad (12)$$

$$i = 1, 2, \ldots, \frac{c}{h} - 1; \quad j = 1, 2, \ldots, \frac{d}{k} - 1,$$

one then obtains for (8, 11)

$$(-\Delta_h + KI) W^{(n+1)} = (KI - P) W^{(n)} + Q, \qquad (13)$$

where P is a diagonal matrix with elements $P_{ij} = p(ih, jk)$, Q is a vector with elements $Q_{ij} = q(ih, jk)$, and I is the identity matrix. The solution of (13) is carried out in each iteration by using a fast direct method.

Finally, under the assumption that the eigenvalues of $(-\Delta_h + KI)^{-1}(KI - P)$ lie in the interval $[-\rho, \rho]$, Chebyshev acceleration is applied:[11]

$$\tilde{W}^{(n+1)} = \omega_{n+1}(W^{(n+1)} - \tilde{W}^{(n-1)}) + W^{(n-1)}, \qquad (14)$$

where $\omega_0 = 1$, $\omega_1 = 2/(2 - \rho^2)$, $\omega_{n+1} = (1 - \rho^2\omega_n/4)^{-1}$ for $n = 1, 2, \ldots,$ and $\tilde{W}^{(n+1)}$ is the improved value of $W^{(n+1)}$, where now $W^{(n+1)}$ satisfies (13) with $W^{(n)}$ replaced by $\tilde{W}^{(n)}$ on the righthand side. This is equivalent to the use in (7) of a sequence $\{\tau_n\}$, rather than a single value of τ, in a manner that is numerically stable and does not require the total number of parameters in the sequence to be specified in advance. If in some cases memory limitations preclude the use of (14), then a fixed sequence $\{\tau_n\}$ could be used instead, ordered in the manner recommended in Reference 12 for numerical stability.

Convergence properties

The convergence properties of the iterative technique can be examined by standard methods in terms of the eigenvalues of the Laplace operator, which are known explicitly for the rectangle. Consider the discrete form of the iteration (7, 8),

$$(-\Delta_h + KI) W^{(n+1)} = (-\Delta_h + KI) W^{(n)} - \tau[(-\Delta_h + P) W^{(n)} - Q], \qquad (15)$$

in which K and τ are not yet specified to be the values (9) and (10). Assume that $K > -\lambda_m$, where λ_m is the smallest eigenvalue of $-\Delta_h$, so that $(-\Delta_h + KI)$ is positive definite. Assume also that the discretization of $\mathfrak{M}$ to $M \equiv -\Delta_h + P$ maintains the positive definiteness. Then one obtains, denoting by ν_m and ν_M the minimum and maximum eigenvalues of the generalized eigenvalue problem

$$M\Phi = \nu(-\Delta_h + KI)\Phi,$$

that the spectral radius ρ for iteration (15) is given by

$$\rho(I - \tau[-\Delta_h + KI]^{-1}M) = \text{Max}(|1 - \tau\nu_m|, |1 - \tau\nu_M|). \qquad (16)$$

Since $\nu_m > 0$, there follows the well-known result[13] that *iteration* (15) *converges for any initial approximation* $W^{(0)}$ *if and only if* $0 < \tau < 2/\nu_M$, *and, for a single parameter* τ, *the optimal choice*

$$\tau = \tau_0 \equiv 2/(\nu_m + \nu_M) \tag{17}$$

yields the smallest spectral radius

$$\rho = \rho_0 \equiv (\nu_M - \nu_m)/(\nu_M + \nu_m). \tag{18}$$

The values of ν_m and ν_M can be estimated from the Rayleigh quotient for ν,

$$\frac{\Phi^T M \Phi}{\Phi^T(-\Delta_h + KI)\Phi} = 1 + \frac{\Phi^T(P - KI)\Phi}{\Phi^T(-\Delta_h + KI)\Phi}. \tag{19}$$

One obtains

$$1 + \min\left(\frac{\beta - K}{\lambda_m + K}, \frac{\beta - K}{\lambda_M + K}\right)$$

$$\leq \nu_m \leq \nu_M \leq 1 + \max\left(\frac{B - K}{\lambda_m + K}, \frac{B - K}{\lambda_M + K}\right), \tag{20}$$

where λ_M is the largest eigenvalue of $-\Delta_h$.

The estimate for ρ obtained from (16) and (20) is least when a choice for K is made such that

$$\beta \leq K \leq B, \tag{21}$$

assuming $\beta > -\lambda_m$ holds. There results that for the corresponding optimal choice

$$\tau = \frac{2(\lambda_m + K)}{2\lambda_m + B + \beta} \tag{22}$$

there holds

$$\rho \leq \rho_u \equiv \frac{B - \beta}{2\lambda_m + B + \beta}. \tag{23}$$

The upper bound (23) on the spectral radius is essentially independent of mesh size, since λ_m is approximately equal to its limiting continuous equivalent of $\pi^2(c^{-2} + d^{-2})$ to order of the square of the mesh length. It is a simple matter to place a rigorous lower bound on λ_m and obtain from (23) the result that $\rho_u < 1$ and hence that convergence is guaranteed, for $\beta > -\lambda_m$.

It is of interest to compare (23) with the analogous spectral radius estimate for the iteration, without scaling and shifting, based on (1). For the latter case one obtains that for the optimal choice $\tau = 2/(\alpha + A)$ there holds

$$\rho \leq (A - \alpha)/(A + \alpha), \tag{24}$$

where $\alpha = \min a(x, y)$ and $A = \text{Max}\, a(x, y)$ on the closed rectangle. The estimate (24) is independent of the mesh size and is the sharpest such one possible.

The presence of the $2\lambda_m$ term in the denominator of (23) can have the effect of there resulting a considerably smaller bound on ρ for the scaled and shifted iteration than results from (24) for iteration (1). Since (24) is essentially sharp such a smaller bound would imply a faster convergence rate. Thus one concludes that scaling and shifting are most effective when A/α is not especially close to one and a does not vary with excessive rapidity over the rectangle, in which case the resulting improvement in convergence rate could be substantial.

Remarks

Lower bound on β

It is required above that β, the minimum of $p(x, y)$ on the rectangle, satisfy $\beta > -\lambda_m$. In the case for which $\beta \leq -\lambda_m$ (the positive definiteness of M does not preclude P dipping below $-\lambda_m$ over a portion of the rectangle) the estimate (20) no longer yields an upper bound on ρ that is less than one, hence it does not guarantee convergence. In the numerical experiments performed on such cases, iteration (15) usually converged, but at a relatively slower rate. In general, the best candidates for the iterative procedure are those cases for which $\beta > -\lambda_m$.

Shift parameter

The choice of the particular value (9) for K out of the possible ones (21) yielding the best convergence rate estimate (23), corresponding to (22), is made for two reasons. One is that for the corresponding value $\tau = 1$, which is obtained from (22) for the shift (9), the resulting discrete Picard iteration (13) requires fewer computer operations than does the one for general τ (15). The other is that for this shift the actual convergence rate observed in numerical experiments is somewhat more rapid than it is for shifts near the end points of the interval $[\beta, B]$, at least for those problems for which $p(x, y)$ varies smoothly without rapid changes (see NUMERICAL EXAMPLES).

Calculation of P

In practice, an alternative to the analytic calculation of $p(x, y) = a^{-1/2}\Delta(a^{1/2})$ and its subsequent numerical evaluation to obtain the elements of P in (13) may be desirable. One could, instead, difference $a^{1/2}(x, y)$

TABLE I—Results after 5 iterations

	a(x,y)	K	τ	Chebyshev Acceleration	ρ_e	Maximum Error
(a)	$[1+\frac{1}{2}(x^4+y^4)]^2$	0	0.868	none	0.13	3.7(−5)
		0	0.868	using ρ_u	—	2.4(−6)
		3	1	none	0.039	3.9(−8)
		3	1	usingρ_u	—	1.1(−6)
		3	1	using ρ_e	—	4.3(−9)
(b)	$[1+\sin\frac{1}{2}\pi(x+y)]^2$	0	16/15	none	0.066	1.2(−6)
		0	16/15	using ρ_u	—	7.2(−8)
		$-\pi^2/8$	1	none	0.061	2.3(−7)
		$-\pi^2/8$	1	using ρ_u	—	3.2(−8)
		$-\pi^2/8$	1	using ρ_e	—	2.3(−8)
(c)	$[2+\tanh 4(x+y-1)]^2$	0	0.829	none	0.31	3.4(−4)
		0	0.829	using ρ_u	—	5.9(−3)
		4.07	1	none	0.26	2.6(−4)
		4.07	1	using ρ_u	—	1.5(−3)
		4.07	1	using ρ_e	—	3.4(−5)

directly to obtain approximate elements p_{ij}^h of P, $p_{ij}^h=\Delta_h a_{ij}^{1/2}/a_{ij}^{1/2}$, where $a_{ij}^{1/2}=[a(ih,jk)]^{1/2}$. The discretization error introduced by using p_{ij}^h instead of $p(ih,jk)$ would be of the same order as that already introduced by (12).

Spectral radius estimate

In applying Chebyshev acceleration (14) to iteration (13), one can either use the estimate (23) for the spectral radius or else obtain an estimate by observing the convergence rate when solving the problem first on a coarse grid. This latter procedure is often worth the small extra expenditure of computing effort, because the estimate (23) may be pessimistic and, since ρ is essentially independent of mesh size, the observed value usually is more accurate.

NUMERICAL EXAMPLES

Well-suited cases

The ideal case for the basic technique (13, 9) is one in which $p=a^{-1/2}\Delta(a^{1/2})$ is constant on the rectangle [e.g., $a=\cos^2(x+y)$, $a=J_0^2([x^2+y^2]^{1/2})$, etc.]. Then from (23) one obtains that the optimal spectral radius is $\rho=0$, hence the problem is solved completely (to round-off accuracy) in only one iteration. This result corresponds to the fact that in each iteration a Helmholtz equation (13) is solved directly.

Other highly suitable cases for the technique are those not departing strongly from the ideal one. The experimental results for two such cases are summarized in Table Ia, b. Both cases were solved numerically by using (15) on the unit square $0<x<1$, $0<y<1$ with uniform mesh spacing $h=k=2^l$, for the values $l=4$, 5, and 6. (The number of rows of interior mesh points should be 2^l-1, l an integer, in at least one direction for fast direct methods to apply efficiently.)

The entries in Table I are the rounded values for a mesh with 64×64 interior points; for the other mesh sizes the values differed from these only slightly, if at all. A value of K equal to 0 or to $(\beta+B)/2$ was used, along with the corresponding value (22) for τ. When Chebyshev acceleration was included, either the estimate ρ_u from (23) or the experimentally observed estimate ρ_e was used to approximate the spectral radius ρ in (14) of $(I-\tau[-\Delta_h+KI]^{-1}[-\Delta_h+P])$. The entries for the value of ρ_e are the observed approximate limiting values of the ratio

$$\|W^{(n)}-W^{(n-1)}\|_\Delta/\|W^{(n-1)}-W^{(n-2)}\|_\Delta,$$

where $\|W\|_\Delta=[W^T(-\Delta_h+KI)W]^{1/2}$. The maximum error, which is listed in the last column, is the maximum of the differences at the mesh points between $W^{(5)}$ and the solution. The initial maximum error had the value of approximately 1.

For the example in Table Ia, $\beta=0$ and $B=6$. Thus the estimate (23) for the optimal spectral radius is $\rho\leq\rho_u\approx 0.132$ (using $2\pi^2$ for λ_m), and the shift (9) is $K=3$. For the example in Table Ib, one has $\beta=-\pi^2/4$,

$B=0$, and $\rho_u \approx 1/15$. In this case, the improvement obtained by using the shift $K=(\beta+B)/2$, instead of $K=0$, is not so great as it is for example Ia.

The effect of scaling and shifting can be found by comparing the results for these two examples with the estimate (24). For both there holds $\alpha=1$ and $A=4$, so that the spectral radius estimate without scaling and shifting in each case is 0.6.

Less well-suited cases

For the example summarized in Table Ic, $p(x, y)$ deviates more strongly from the ideal case. The task of calculating the actual extremal values of $p(x, y)$ on $\mathcal{R}$ was not carried out for this example; instead, the discrete equivalents $\beta=\beta_h=\min P_{ij}$, $B=B_h=\max P_{ij}$ were used. For the 64×64 mesh, $\beta_h \approx -9.62$ and $B_h \approx 17.77$, for which $\rho_u \approx 0.575$. Note that here $K=0$ does not correspond to an end point of the interval $[\beta, B]$.

An investigation of the possible non-sharpness of estimate (20) and non-optimality of (9) and (10), which are more important here than in a nearly ideal case, was carried out by fixing τ at the value one and observing the change in ρ_e as K was varied. A local minimum was found at approximately $K=3.0$, for which ρ_e is approximately 0.23.

For the more extreme case

$$a(x, y)=[2+\tanh 10(x+y-1)]^2,$$

in which the change in the value of a in crossing the line $x+y=1$ is very abrupt, β_h and B_h becomes approximately -60 and 111, respectively. In this case $\beta < -\lambda_m$; hence, the estimate (23) yields merely that $\rho \leq \rho_u > 1$. The iteration did converge, however, with the observed spectral radius $\rho_e \approx 0.63$ and a maximum error of 2.5×10^{-2} after five iterations for the usual test problem, with $K=(\beta_h+B_h)/2$ and $\tau=1$. With the inclusion of Chebyshev acceleration based on this value of ρ_e, the maximum error after five iterations was reduced to 6.3×10^{-3}. The value of ρ_e can be decreased in this case, with τ fixed at 1, to a locally minimum value of approximately 0.54 at approximately $K=14$.

Computational requirements

All the above experiments were carried out using the subroutine BUXYDY, written by B. L. Buzbee at Los Alamos Scientific Laboratory, which solves the Helmholtz equation on a rectangle using Buneman's algorithm for odd-even reduction.[4] The subroutine requires approximately 0.06 seconds on the CDC 7600 computer to solve a problem on a 64×64 mesh.

Qualitative comparison of the computational requirements of the technique with those of other methods can be made using the operation-count table given in Reference 3. One finds, for example, that for a 64×64 mesh the operations required for one iteration of (13) are equivalent to those required for about 4 or 4½ SOR iterations, and that about 85 SOR iterations are required to reduce the initial error by a factor $N^{-2} \approx 2.5\times10^{-4}$ (discretization error order) in the numerical solution of the Poisson equation when optimal parameters are used. The solution of (3) or (6) by SOR would generally require even more iterations.

The memory requirements of (13, 14) exceed those of SOR by about $3N^2$ locations if both $P-KI$ and $\tilde{W}^{(n-1)}$ are stored. This value can be reduced to N^2, however, in exchange for recomputing $P-KI$ at each iteration and using a form of Chebyshev acceleration that requires, instead of $\tilde{W}^{(n-1)}$, a sequence of parameters $\{\tau_n\}$.

One concludes that for well-suited cases, such as those in Table Ia, b, the basic technique is an extremely efficient one and compares very favorably with standard iterative and elimination methods. Its advantages are especially striking for problems with a large number of mesh points. For less well-suited problems, the technique may be very satisfactory in some cases, but further study would be helpful to clarify the best means for estimating the parameters.

EXTENSIONS

The iterative technique can be modified to handle more general equations and boundary conditions than those discussed here and to solve problems that are discretized on a mesh with non-uniform spacing.[14]

ACKNOWLEDGMENTS

This study was performed in collaboration with Gene H. Golub and is reported under our joint authorship in a larger paper, which includes further elaboration of many of the points discussed here.[14]

REFERENCES

1 O BUNEMAN
A compact non-iterative Poisson solver
Report 294 Stanford University Institute for Plasma Research Stanford California 1969

2 R W HOCKNEY
The potential calculation and some applications
Methods in Computational Physics vol 9 B Adler S Fernbach and M Rotenberg eds Academic Press New York and London 1969 pp 136–211

3 F W DORR
The direct solution of the discrete Poisson equation on a rectangle
SIAM Rev 12 1970 pp 248–263

4 B L BUZBEE G H GOLUB C W NIELSON
On direct methods for solving Poisson's equations
SIAM J Numer Anal 7 1970 pp 627–656

5 E G D'YAKONOV
On an iterative method for the solution of finite difference equations
Dokl Akad Nauk SSSR 138 1961 pp 522–525

6 O B WIDLUND
On the use of fast methods for separable finite difference equations for the solution of general elliptic problems
Sparse Matrices and Applications D J Rose and R A Willoughby eds Plenum Press New York 1972 pp 121–134

7 J E GUNN
The numerical solution of $\nabla a \nabla u = f$ by a semiexplicit alternating direction iterative method
Numer Math 6 1964 pp 181–184

8 J E GUNN
The solution of elliptic difference equations by semiexplicit iterative techniques
SIAM J Numer Anal 1 1965 pp 24–25

9 H L STONE
Iterative solution of implicit approximations of multidimensional partial differential equations
SIAM J Numer Anal 5 1968 pp 530–558

10 T DUPONT R P KENDALL H H RACHFORD JR
An approximate factorization procedure for solving self-adjoint elliptic difference equations
SIAM J Numeral Anal 5 1968 pp 559–573

11 R S VARGA
Matrix iterative analysis
Prentice-Hall Englewood Cliffs New Jersey 1962 p 141 prob 8

12 V I LEBEDEV S A FINOGENOV
On the order of choice of the iteration parameters in the Chebyshev cyclic iteration method
Zhur Vych Mat i Mat Fiz 11 1971 pp 425–438 English translation in Report CS72-304 Computer Science Department Stanford University Stanford California 1972

13 E L STIEFEL
Über einige methoden der relaxationsrechnung
Z angew Math Phys 3 1952 pp 1–33

14 P CONCUS G H GOLUB
Use of fast direct methods for the efficient numerical solution of nonseparable elliptic equations
To appear. Also available as Report 72-278
Computer Science Dept Stanford University
Stanford California 1972

Computer oriented methods for fitting tabular data in the linear and nonlinear least squares sense*

by K. M. BROWN**

University of Minnesota
Minneapolis, Minnesota

INTRODUCTION

Let the tabular data (t_i, y_i), $i = 1, \ldots, m$ be given along with a function (form), say $g(a_1, \ldots, a_n; t)$, which is to be used to fit the data. We wish to determine the parameters a_i in such a way as to

$$\text{minimize } \phi(a_1, \ldots, a_n) \equiv \sum_{i=1}^{m} [g(a_1, \ldots, a_n; t_i) - y_i]^2 \tag{1}$$

This type of approach is called fitting the data in the least squares sense. If g is a linear (nonlinear) function of the parameters a_i, we speak of (1) as a linear (nonlinear) least squares problem.

LINEAR LEAST SQUARES PROBLEMS

In this section we will discuss three techniques for determining the a_i which appear in (1) when these parameters enter g in a linear fashion. Let

$$\mathbf{x} \equiv (x_1, x_2, \ \ldots, x_N)^T$$

and

$$\| \mathbf{x} \| \equiv \left(\sum_{i=1}^{N} x_i^2 \right)^{1/2}.$$

For linear problems (1) can be written as

$$\text{minimize } \| M\mathbf{a} - \mathbf{y} \|^2 \tag{2}$$

where M is a matrix which can depend on $\mathbf{t}$ but does not depend on $\mathbf{a}$.

Example. When fitting tabular data with a polynomial function

$$g(a_1, \ \ldots, a_n; t) = a_n t^{n-1} + a_{n-1} t^{n-2} + \cdots + a_2 t + a_1,$$

the ith row of the matrix M is simply

$$1 \quad t_i \quad t_i^2 \quad \ldots \quad t_i^{n-2} \quad t_i^{n-1}$$

The first technique: The classical, but wrong way to proceed—unfortunately enjoying wide usage today!

Relying on a well-known theorem from the calculus which states that the points which minimize ϕ in (1) must be contained among the zeros of the partial derivative system $\partial\phi/\partial a_i = 0$, $i = 1, \ldots, n$, it follows that the minimum of ϕ in the linear case (2) is obtained by solving the linear system of equations (the *normal equations*)

$$M^T M\mathbf{a} = M^T\mathbf{y} \tag{3}$$

Example. With reference to the previous example of polynomial fitting, the ith, jth element of the coefficient matrix M^TM in (3) is given by $\sum_{k=1}^{m} t_k^{i+j-2}$ and the ith element of the right hand side $M^T\mathbf{y}$ of (3) is simply $\sum_{k=1}^{m} t_k^{i-1} y_k$. The difficulty with this seemingly natural (and mathematically exact!) approach is that the system of equations (3) can be highly *ill-conditioned.* This means that slight changes in the coefficient matrix M^TM produce enormous changes in the solution $\mathbf{a}$. In 1957 George Forsythe[1] showed that by using orthogonal polynomials for the polynomial fitting problem, the amount of ill-conditioning associated with (3) could be reduced considerably. The Gram-Schmidt process is usually used to generate the orthogonal polynomials. This process has the advantage of not having to start from scratch when going from a polynomial fit of degree $n-1$ to one of degree n. A number

* This work was supported in part by the National Science Foundation under Grant GJ-32552.

** This paper is dedicated to the memories of George E. Forsythe and Hirondo Kuki.

of the better subroutine libraries contain a data fitting program using Gram-Schmidt produced orthogonal polynomials. Unfortunately this procedure does date back to 1957 and still requires the explicit formation of the coefficient matrix M^TM of (3).

Example. The formation of M^TM can result in a loss of numerical accuracy or even introduce an artificial singularity. Let (Golub and Reinsch[2])

$$M=\begin{bmatrix}1 & 1\\ e & 0\\ 0 & e\end{bmatrix}$$

then

$$M^TM=\begin{bmatrix}1+e^2 & 1\\ 1 & 1+e^2\end{bmatrix}.$$

If $e^2<t$, the machine tolerance, then M^TM appears as the singular matrix consisting of all ones in the computer and it is impossible to solve the normal equations (3).

On the other hand, as we shall see in the next two sections, recent techniques actually permit us to solve the minimization problem (2) *without explicitly having to form the matrix M^TM of* (3)*!*

The second technique: The Businger-Golub procedure for matrices with rank n

If the $m\times n$ matrix M of (2) is such that $m\geq n$ and rank $(M)=n$, a very efficient procedure due to Businger and Golub[3] can be used to solve (2) without ever having to form M^TM explicitly. The procedure makes use of the fact that the euclidean norm $\|\cdots\|$ is invariant under unitary transformations, so that

$$\|M\mathbf{a}-\mathbf{y}\|=\|UM\mathbf{a}-\mathbf{Z}\|$$

where $\mathbf{Z}=U\mathbf{y}$ and $U^TU=I$. Now U can be chosen so that

$$UM=T=\begin{bmatrix}\tilde{T}\\ \cdots\\ 0\end{bmatrix}. \qquad (4)$$

where $\tilde{T}$ is an $n\times n$ upper triangular matrix and the matrix 0 is of size $(m-n)\times n$. The point $\mathbf{a}^*$ which minimizes (2) is then given by $\mathbf{a}^*=\tilde{T}^{-1}\tilde{\mathbf{Z}}$, where $\tilde{\mathbf{Z}}$ denotes the first n components of $\mathbf{Z}$. The procedure uses Householder transformations to carry out the decomposition (4). These transformations are extremely stable and the computations necessitated are quite efficient. A bonus provided by the decomposition (4) is the ease of calculating $(M^TM)^{-1}$ and $\det(M^TM)$, quantities necessary in many statistical calculations. As pointed out by Businger and Golub[3] page 274, since $(M^TM)=\tilde{T}^T\tilde{T}$, then $(M^TM)^{-1}=\tilde{T}^{-1}(\tilde{T}^T)^{-1}$; moreover, $\det(M^TM)=(\det\tilde{T})^2=(t_{11}\cdot t_{22}\ldots t_{nn})^2$.

The FORTRAN IV subroutine LSTSQ which implements this procedure was obtained from Dr. P. A. Businger, Bell Telephone Labs., Murray Hill, New Jersey 07974. I tested the routine on a number of examples and the behavior was excellent (as long as rank $(A)=n$). Probably the outstanding feature of the program (i.e., of the method) is its compactness: LSTSQ has only 40 FORTRAN statements!

Users of APL may be interested in knowing that whenever they invoke the "domino" function (an APL primitive operator) to solve a linear least squares problem, they are using the Businger-Golub procedure[3] as modified by Jenkins[4] to include the interchange and scaling strategies suggested by Powell and Reid.[5]

Remark. Another excellent (competitive) method is due to Björck[6] and we are in the process of testing it vs. LSTSQ.

The third technique: The singular value decomposition approach of Golub and Reinsch which applies whether or not M has full rank

In a number of important problems it can happen that M does not have full rank; i.e., rank $(M)<n$. The techniques of the previous sections break down in this case; however, Golub and Reinsch[2] in 1970 gave an effective numerical procedure for this problem. Again, their technique avoids the explicit formation of M^TM. Their approach is based on the fact that an $m\times n$ matrix M with $m\geq n$ can be decomposed

as
$$M=U\Sigma V^T \qquad (5)$$

where $U^TU=V^TV=VV^T=I_n$ and $\Sigma=\text{diag}(\sigma_1,\ldots,\sigma_n)$. The matrix U is made up of the (orthonormalized) eigenvectors associated with the n largest eigenvalues of MM^T, and the matrix V is made up of the (orthonormalized) eigenvectors of M^TM. The σ_i are the nonnegative square roots of the eigenvalues of M^TM and are called *singular values.* The decomposition (5) is called the *singular value decomposition* (SVD) of M. It can be shown that the point $\mathbf{a}^*$ which minimizes (2) is given by

$$\mathbf{a}^*=V\Sigma^+U^T\mathbf{y}$$

whether or not the rank of M is less than n; of course if rank $(M)<n$ the solution $\mathbf{a}^*$ is *not unique.* Here Σ^+ is the pseudoinverse of the matrix Σ and is given ex-

plicitly by

$$\Sigma^+ = \text{diag } (\sigma_i^+)$$

where $$\sigma_i^+ = \begin{cases} 1/\sigma_i & \text{for } \sigma_i > 0 \\ 0 & \text{for } \sigma_i = 0. \end{cases}$$

In order to produce the singular value decomposition of M, the Golub-Reinsch[2] technique uses Householder transformations to reduce M to a bidiagonal form and then uses the QR algorithm to solve for the singular values of the resulting bidiagonal matrix.

As Golub has pointed out to us in private communication, there is again no difficulty in obtaining from (5) the quantity $(M^TM)^{-1}$ (of interest in statistical work), when it exists, since

$$(M^TM)^{-1} = V\Sigma^{-2}V^T$$

where $$\Sigma^{-2} = \text{diag } (\sigma_1^{-2}, \sigma_2^{-2}, \ldots, \sigma_n^{-2}).$$

A single precision FORTRAN IV subroutine SVD was obtained from Dr. P. A. Businger (address given above); its double precision analog DSVD was obtained from Professor Gene H. Golub, Department of Computer Science, Stanford University, Stanford, California 94305. A complex-arithmetic FORTRAN IV version CSVD has appeared as a CACM algorithm.[7] I have tested SVD extensively and am happy to report that it gave excellent results. Unfortunately one has to pay a price to implement an algorithm that does not require M to have full rank, namely, the price of length of program: SVD consists of approximately 230 FORTRAN statements vs. only 40 for LSTSQ. Of course SVD furnishes much more information about the problem than LSTSQ does; thus SVD calculates the rank of M and the eigenvalues and eigenvectors of A^TA, etc.

Some conclusions about linear least squares data fitting

If one is only trying to solve (2) and the matrix M is known to be of rank n, then LSTSQ should be used. On the other hand if the rank of M is unknown, or rank $(M) < n$, or additional information such as eigenvalues and eigenvectors are sought SVD should be used. It is of interest to note that for problems having full rank no difference in accuracy was noted between LSTSQ and SVD.

NONLINEAR LEAST SQUARES PROBLEMS

We now return to the consideration of problem (1) for the case when the fitting function (form) g is a nonlinear function of its parameters a_i. Let us write

$$f_i = g(a_1, \ldots, a_n; t_i) - y_i, \quad i = 1, \ldots, m,$$

and let $F = (f_1, \ldots, f_m)^T$, so that we wish to minimize $\sum_{i=1}^m f_i^2$ or minimize $\| F \|^2$.

Until the late 1960's, I believed along with many of my colleagues that the Davidon-Fletcher-Powell[8,9] algorithm was the soundest computational approach to this problem. It was a great eye-opener, then, to see Yonathan Bard's paper[10] in the spring of 1970, which concluded after a very careful and exhaustive set of tests that modifications of the Gauss method[11] (including the Levenberg-Marquardt method)[12,13] performed best in practice, followed by variable metric rank one methods, *followed by* Davidon-Fletcher-Powell[8,9] methods. Bard raised the question of how the finite-difference (derivative-free) analogs of methods like the Levenberg-Marquardt algorithm[12] would perform. We will examine some newer algorithms for the nonlinear least squares problem and report the results of recent computational experience with these methods. One of these methods provides an answer (in the affirmative) to the question raised by Bard. Finally we will look at a most interesting new class of methods for nonlinear problems whose variables separate.

The general strategy: Find the zeros of the gradient system

It is known from the calculus of functions of several variables that the points which are relative minima of $\| F \|^2$ are contained among the zeros of the gradient of ϕ, written $\nabla\phi(\mathbf{a})$. The gradient is a vector of length n whose ith component is $\partial\phi/\partial a_i$. It is easy to verify that

$$\nabla\phi(\mathbf{a}) = 2J_F^T(\mathbf{a}) \cdot F(\mathbf{a}),$$

where $J_F(\mathbf{a})$ denotes the $m \times n$ Jacobian matrix of F. Define $G(\mathbf{a}) \equiv J_F^T(\mathbf{a}) \cdot F(\mathbf{a})$. We seek the zeros of $G(\mathbf{a})$.

Newton's method as a model

Since Newton's method (or a Newton-like method) is so effective in solving nonlinear systems of equations, it seems natural to try to solve $G(\mathbf{a}) = 0$ by Newton's method. The iteration (given a starting guess $\mathbf{a}^0$) is given by

$$\mathbf{a}^{l+1} = \mathbf{a}^l - J_G(\mathbf{a}^l) \cdot G(\mathbf{a}^l), \quad l = 0, 1, 2, \ldots.$$

Now $J_G(\mathbf{a})$ is given by

$$J_G(\mathbf{a}) = \sum_{k=1}^m f_k(\mathbf{a}) \cdot H_k(\mathbf{a}) + J_F(\mathbf{a})^T J_F(\mathbf{a}),$$

where $H_k(\mathbf{a})$ is the Hessian matrix of f_k at $\mathbf{a}$; i.e., the

ith, jth element of $H_k(\mathbf{a})$ is given by $\partial^2 f_k(\mathbf{a})/\partial a_j\, \partial a_i$. Thus Newton's method, applied to solving $G(\mathbf{a})=0$ is given by the following

$$\mathbf{a}^{l+1}=\mathbf{a}^l-\left[\sum_{k=1}^{m} f_k(\mathbf{a}^l)\cdot H_k(\mathbf{a}^l)+J_F(\mathbf{a}^l)^T J_F(\mathbf{a}^l)\right]^{-1} J_F(\mathbf{a}^l)^T F(\mathbf{a}^l) \qquad (6)$$

The latter formula requires (assuming continuous second partial derivatives of ϕ) the calculation of $M\cdot N\cdot(N+1)/2$ second partial derivatives per iteration. For even mildly complicated functions, f_k, this can be a horrendous and error-ridden calculation. The Gauss[11] and Levenberg-Marquardt[12,13] methods are two frequently used approaches to avoid the calculation of the Hessians. Gauss' method simply drops the matrix $\sum_{k=1}^{m} f_k H_k$ in (6) whereas the Levenberg-Marquardt algorithm approximates that matrix by a scalar matrix $\mu^l I$. Our experience is that both of these methods behave well local to a zero, $\mathbf{a}^*$, of G, *provided that* $\| F(\mathbf{a}^*) \|$ is very small relative to the magnitudes of $\| H_k \|$, $k=1, \ldots, m$ and $\| J_F^T(\mathbf{a}^*)\cdot J_F(\mathbf{a}^*) \|$. The following methods address Bard's question.

Derivative free analogs of the Levenberg-Marquardt and Gauss algorithms for nonlinear least squares approximation

Recently Brown and Dennis[14] have shown that there is no degradation in the theoretical or actual behavior of the Gauss and Levenberg-Marquardt algorithms when these algorithms are discretized (actual derivatives being replaced by first differences) in a particular manner. We detail the algorithm as follows:
Given $\mathbf{a}^0$, form successively

$$\mathbf{a}^{l+1}=\mathbf{a}^l+\Delta\mathbf{a}^l, \quad l=0, 1, 2, \ldots,$$

where $\Delta\mathbf{a}^l$ is obtained by using Cholesky's method to solve the positive-definite symmetric linear system

$$[\mu^l I+\tilde{J}^{l^T}\tilde{J}^l]\cdot\Delta\mathbf{a}^l=\tilde{J}^{l^T}F(\mathbf{a}^l). \qquad (7)$$

Here the ith, jth element of $\tilde{J}^l$ is given by

$$\frac{f_i(\mathbf{a}^l+h_j^l\mathbf{u}_j)-f_i(\mathbf{a}^l)}{h_j^l}$$

where $\mathbf{u}_j$ denotes the $\mathbf{j}$th unit column vector and

where

$$h_j^0=\min\{\| F(\mathbf{a}^0) \|_{\text{any}}, \delta_j^0\}$$

and

$$h_j^l=\min\{\| F(\mathbf{a}^l) \|_{\text{any}}, \delta_j^l, \| \mathbf{a}^l-\mathbf{a}^{l-1} \|_{\text{any}}\}, \quad l>0$$

and where

$$\delta_j^l=\begin{cases} 10^{-\beta} & \text{if } | a_j^l |<10^{-\beta+3} \\ .001\times| a_j^l | & \text{if } 10^{-\beta+3}\leq| a_j^l |, \end{cases}$$

with $2\times\beta$ being the machine tolerance.

Finally if in (7) $\mu^l=0$ for all l, we have the *derivative-free Gauss' method*, whereas if $\mu^l=c\cdot\| F(\mathbf{a}^l) \|_{\text{any}}$, we have the *derivative-free Levenberg-Marquardt method.* The techniques are derivative-free in the full sense of that phrase: the only things that the scientist has to provide are his functions f_k. He doesn't even have to furnish initial estimates to any of the derivatives.

Remark. We note that if in (7) $\mu^l\rightarrow\infty$ we have a pure gradient (or descent) method, whereas if $\mu^l\rightarrow 0$ and J is square we get exactly the discrete Newton's method. This is why the Levenberg-Marquardt method has been called "Marquardt's compromise." Now Levenberg[12] has shown that his method is always norm-reducing, i.e., $\| F(\mathbf{a}^{l+1}) \|<\| F(\mathbf{a}^l) \|$, provided that μ^l is chosen sufficiently large. Thus the Levenberg-Marquardt algorithm has global stability properties not held by Gauss' method. On the other hand, close to a root Gauss' method can converge quite rapidly—as Newton's method does. This makes the strategy clear when choosing c above to define μ^l: when one is far away from the solution, choose c large so as to weight the descent part of the correction and keep the iteration stable. As the solution is approached, i.e., as $\| G(\mathbf{a}) \|$ becomes small, decrease c so as to enhance the Gauss part of the correction and produce more rapid convergence. In 14 we have proven that the derivative-free Levenberg-Marquardt and Gauss methods converge locally and that the convergence rate is second order (quadratic) whenever $\| F(\mathbf{a}^*) \|=0$. We give a number of numerical results, all of which show that the derivative-free methods behave almost identically to their original analytic counterparts: when convergence occurred both techniques converged at the same rate. As an example consider the norm of the error vectors produced by two of the methods.[14, page 296]

Derivative Free	
Levenberg-Marquardt	Levenberg-Marquardt
7.70 E—01	7.71 E—01
7.07 E—01	7.08 E—01
3.16 E—01	3.17 E—01
2.76 E—02	2.78 E—02
2.03 E—04	2.10 E—04
1.27 E—08	1.43 E—08

Qualitatively the methods behaved the same. This example also showed the quadratic convergence rate predicted by the theory.

A FORTRAN IV subroutine FDLM is available from the author. This technique is also included in IMSL's Library 1 as subroutine ZXMARQ. Information about this truly excellent (commercial) library of over 200 mathematical and statistical FORTRAN routines can be obtained from Dr. Olin G. Johnson, Director of Mathematics, International Mathematical and Statistical Libraries, Inc., 6200 Hillcroft, Houston, Texas 77036.

Reminder. As we indicated just before this section, these techniques work well only when $\| F(\mathbf{a}^*) \|$ is sufficiently small; translation: these are good techniques to use if your form, g, can (by proper parameter estimation—which these algorithms do automatically) be made to provide a good fit to your data.

The large residual problem and an algorithm for attacking it

In a multitude of real world problems $\| F(\mathbf{a}) \|$ turns out to be large even at the minimum $\mathbf{a}^*$. In that case dropping the term $\Sigma f_k H_k$ in (6) or replacing it with just a diagonal matrix yields a poor approximation to the true $J_G(\mathbf{a})$. Hence, Brown and Dennis[15] proposed an algorithm which approximates the initial Hessians and then uses a Powell[16] updating technique to continue the approximations for $l=1, 2, \ldots$. This is *not* a derivative-free technique: the scientist must furnish his functions f_k *and* all *first* partial derivatives, $\partial f_k/\partial a_j$, of those functions. The technique does not require that the scientist furnish any second partial derivatives since these are automatically approximated and updated by the method. We detail the algorithm below.

Algorithm. Let $\mathbf{a}^l$, $J_F(\mathbf{a}^l)$ and $F(\mathbf{a}^l)$ be given along with M matrices $B_{1,l}, \ldots, B_{m,l}$ each of size $n\times n$. (Initially the $B_{i,0}$ may be chosen to approximate the $H_i(\mathbf{a}^0)$ by, say, using first forward differences on the entries of $J_F(\mathbf{a})$; this technique was used in the numerical experiments reported below.) Obtain

$$\mathbf{a}^{l+1}=\mathbf{a}^l-\left[\sum_{k=1}^{m} f_k(\mathbf{a}^l)B_{k,l}+J_F{}^T(\mathbf{a}^l)\cdot J_F(\mathbf{a}^l)\right]^{-1}G(\mathbf{a}^l) \quad (8)$$

and compute $J_F(\mathbf{a}^{l+1})$ and $F(\mathbf{a}^{l+1})$. Now update the B_i by means of Powell's[16] symmetric form of Broyden's[17] "single-rank" updating technique

$$\begin{aligned} B_{i,l+1}=B_{i,l}+[\nabla f_i(\mathbf{a}^{l+1})^T-\nabla f_i(\mathbf{a}^l)^T \\ -B_{i,l}\Delta\mathbf{a}^l]\Delta\mathbf{a}^{lT}/\| \Delta\mathbf{a}^l \|^2+\frac{\Delta\mathbf{a}^l}{\| \Delta\mathbf{a}^l \|^2} \\ \times[\nabla f_i(\mathbf{a}^{l+1})^T-\nabla f_i(\mathbf{a}^l)^T-B_{i,l}\nabla\mathbf{a}^l]^T \quad (9) \\ -\Delta\mathbf{a}^l[\nabla f_i(\mathbf{a}^{l+1})^T-\nabla f_i(\mathbf{a}^l)^T-B_{i,l}\Delta\mathbf{a}^l]^T \\ \times\Delta\mathbf{a}^l\,\frac{\Delta\mathbf{a}^{lT}}{\| \Delta\mathbf{a}^l \|^4}. \end{aligned}$$

Here $\Delta\mathbf{a}^l\equiv\mathbf{a}^{l+1}-\mathbf{a}^l$ and $\nabla f_i(\mathbf{a})$ is just the ith row of $J_F(\mathbf{a})$. Note that the algorithm (8)-(9) requires no more function and derivative evaluations per iterative step than do the Gauss and Levenberg-Marquardt methods. Obviously, more storage is required and there is the extra arithmetic necessary to update the $B_{i,l}$'s. It is necessary that the initial $B_{i,0}$'s be symmetric in order to effect the storage savings possible when using the Powell symmetric update procedure. Convergence theorems for this method are given by Brown and Dennis[15] and the authors have recently established the superlinear convergence of the method.

Numerical results

We wish to minimize $\| F(\mathbf{a}) \|^2$ where $\mathbf{a}=(a_1, a_2, a_3, a_4)$ and F is given by

$$\begin{aligned} f_1(\mathbf{a})&=a_1{}^2\times(a_2-1)^2+50 \\ f_2(\mathbf{a})&=(a_2-1)\times a_3 \\ f_3(\mathbf{a})&=(a_3-5)\times a_4 \\ f_4(\mathbf{a})&=(a_4-10)^2+50 \\ f_5(\mathbf{a})&=a_1{}^2+(a_2-1)^2+(a_3-5)^2+10 \\ f_6(\mathbf{a})&=(a_4-10)^2+(a_3-5)\times(a_2-1). \end{aligned}$$

The minimum $\mathbf{a}^*=(0, 1, 5, 10)$ and the residual at the minimum is large, $\| F(\mathbf{a}^*) \|^2=5100$. As a first experiment $\mathbf{a}^0$ was chosen very close to the root to check out the local convergence properties of the method; $\mathbf{a}^0=(.01, 1.01, 5.01, 10.01)$. The Brown-Dennis algorithm[15] converged after two iterations to an accuracy of $\| \mathbf{a}^2-\mathbf{a}^* \|=1.47\times10^{-8}$ and $\| G(\mathbf{a}^2) \|^2=9.9\times10^{-14}$. The number of evaluations used was 31; however, 16 of these 31 evaluations were used to approximate the initial Hessians.

The Levenberg-Marquardt[12,13] algorithm failed to converge on this example in 200 iterations when started from that same (rather good) starting guess. The difficulty was in getting the fourth component a_4 to settle down. The best value produced gave $\| \mathbf{a}^{97}-\mathbf{a}^* \|=2.68\times10^{-3}$.

For the guess $\mathbf{a}^0=(2.5, 4.5, 10.75, 21)$, the Brown-Dennis method[15] produced convergence in 30 iterations (171 evaluations) to an accuracy of $\| \mathbf{a}^{30}-\mathbf{a}^* \|=4.16\times10^{-7}$ with $\| G(\mathbf{a}^{30}) \|^2=3.05\times10^{-11}$. Again the Levenberg-Marquardt[12,13] method did not converge in 200 iterations.

When a starting guess quite far from the solution was tried, $\mathbf{a}^0=(7.5, 14.5, 35.75, 71)$, a strange thing happened. Even though $\mathbf{a}^{34}=(.80, 2.97, 4.12, 8.51)$ which was well within the sphere centered at the root and containing the second starting guess, the method failed to improve very much during the next 160 iterations! The reason for this is that the approximate Hessians never have a chance to recover from their initial values which were so far removed from $H_k(\mathbf{a}^*)$. An examination of the updating formula (9) reveals that even $B_{i,200}$ contains information propagated from $B_{i,0}$. Thus the $B_{i,l}$ should be restarted (perhaps periodically) from scratch, say whenever $\| \mathbf{a}^{l+1} - \mathbf{a}^l \| < 10^{-p}$, where p is incremented from 0 to 1, etc.

A copy of the FORTRAN IV subroutine NLSQ which implements the Brown-Dennis algorithm[15] is available from the author.

A hybrid procedure

Testing is currently being conducted on a hybrid technique which uses the Levenberg-Marquardt[12,13] algorithm until two successive iterates agree to within one significant digit and then a switchover is effected to the Brown-Dennis method.[15] Initial results look encouraging.

Spline regression

For the scientist who wishes to curve fit for purposes of prediction or interpolation and who does *not* have a specific form (model), g, in mind as dictated by, say, natural physical or biological laws, cubic splines are an excellent choice to use as the fitting functions. Greville,[18] Nilson[19] or Schultz[20] have defined and given properties of these functions and Greville[18] and Nilson[19] include some specific computer oriented algorithms for spline computations. Based upon some recent computational experience, this author recommends specifically the piecewise cubic Hermite polynomials as the fitting functions.

Again the IMSL Library (see above) contains a number of routines for data fitting using cubic splines.

Nonlinear least squares problems whose variables separate

Example. In most of the nonlinear least squares problems I have been asked to look at, the scientist has a form, g, in mind like the following

$$g(a_1, \ldots, a_6; t) = a_1 + a_2 t + a_3 e^{-a_4 t} + a_5 \sin a_6 t,$$

the point being that about half of his parameters enter the form in a linear fashion. Now we can certainly solve this problem (viewed strictly as a six parameter nonlinear problem) using the techniques of the previous sections; however, something tells us intuitively that there should be some way of isolating the linear parameters a_1, a_2, a_3 and a_5 from this problem. More generally consider the function, g, of (1) to have the form

$$g(\mathbf{a}, \boldsymbol{\alpha}; t) = \sum_{j=1}^{n} h_j(\mathbf{a}) \psi_j(\boldsymbol{\alpha}; t),$$

where $\mathbf{a} = (a_1, \ldots, a_s)$ and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k)$. Scolnik[21] has shown that indeed it is possible to optimize first with respect to some of the parameters and then later with respect to the rest. More recently Golub and Pereyra[22] have generalized Scolnik's results. Their paper[22] contains FORTRAN programs for this important class of nonlinear least squares problems.

SUMMARY

We have tried to present some of the best computer oriented techniques for the linear and nonlinear least squares problems. For the linear problem we have recommended reduction by Householder transformations in the full rank case and using the singular value decomposition in other cases. For nonlinear problems we recommended the derivative free Levenberg-Marquardt method for problems having small residuals and the Brown-Dennis technique (with restarts made periodically to the approximate Hessians). For nonlinear problems whose variables separate, there is an important new approach due to Scolnik, Golub and Pereyra. Finally when no model is known, *a priori*, spline regression was recommended.

ACKNOWLEDGMENTS

The author is grateful to Dr. Peter A. Businger and Professor Gene H. Golub for their helpful discussion relative to portions of this paper and for their furnishing the FORTRAN programs LSTSQ, SVD, DSVD and CSVD.

REFERENCES

1 G E FORSYTHE
Generation and use of orthogonal polynomials for fitting data with a digital computer
J Soc Indust Appl Math 5 pp 74-88 1957

2 G H GOLUB C REINSCH
Singular value decomposition and least squares solutions
Numer Math 14 pp 403-420 1970

3 P A BUSINGER G H GOLUB
Linear least squares solutions by Householder transformations
Numer Math 7 pp 269-276 1965

4 M A JENKINS
The solution of linear systems of equations and linear least squares problems in APL
IBM New York Scientific Center Technical Report
No 320-2989 1970

5 M J D POWELL J K REID
On applying Householder transformations to linear least squares problems
Proceedings of the IFIP Congress in Edinburgh 1968

6 A BJORCK
Solving least squares problems by Gram-Schmidt orthogonalization
BIT 7 pp 1-21 1967

7 P A BUSINGER G H GOLUB
Algorithm 358—Singular value decomposition of a complex matrix
CACM 12 pp 564-565 1969

8 W C DAVIDON
Variable metric method for minimization
AEC Research and Development Report ANL-5990 (Rev) 1959

9 R FLETCHER M J D POWELL
A rapidly convergent descent method for minimization
Comput J 6 pp 163-168 1963

10 Y BARD
Comparison of gradient methods for the solution of nonlinear parameter estimation problems
SIAM J on Num Anal 7 pp 157-186 1970

11 K F GAUSS
Theoria Motus Corporum Coelistiam
Werke 7 pp 240-254 1809

12 K LEVENBERG
A method for the solution of certain nonlinear problems in least squares
Quart Appl Math 2 pp 164-168 1944

13 D W MARQUARDT
An algorithm for least squares estimation of nonlinear parameters
SIAM J on Num Anal 2 pp 431-441 1963

14 K M BROWN J E DENNIS JR
Derivative free analogues of the Levenberg-Marquardt and Gauss algorithms for nonlinear least squares approximation
Numer Math 18 pp 289-297 1972

15 K M BROWN J E DENNIS JR
New computational algorithms for minimizing a sum of squares of nonlinear functions
Yale University Dept of Computer Science Res Rpt No 71-6 1971

16 M J D POWELL
A new algorithm for unconstrained optimization
Nonlinear programming J B Rosen O L Mangasarian and K Ritter (eds) Academic Press New York pp 31-65 1970

17 C G BROYDEN
The convergence of single-rank Quasi-Newton methods
Math Comp 24 pp 365-382 1970

18 T N E GREVILLE
Spline functions, interpolation and numerical quadrature
Mathematical Methods for Digital Computers Vol 2
A Ralston and H S Wilf (eds) Wiley New York pp 156-168 1967

19 E N NILSON
Cubic splines on uniform meshes
CACM 13 pp 255-258 1970

20 M H SCHULTZ
Multivariate spline functions and elliptic problems
Approximations with Special Emphasis on Spline Functions Academic Press New York pp 279-347 1969

21 H D SCOLNIK
On the solution of nonlinear least squares problems
Proc IFIP-71 pp 18-23 1971

22 G H GOLUB V PEREYRA
The differentiation of pseudoinverses and nonlinear least squares problems whose variables separate
Stanford University Computer Science Department Tech Rpt STAN-CS-72-261 1972

An efficient band-oriented scheme for solving n by n grid problems

by ALAN GEORGE

University of Waterloo
Waterloo, Ontario, Canada

INTRODUCTION

Let $N=n^2$ for some positive integer n, and consider a square n by n grid consisting of $(n-1)^2$ small squares and having a node at each of the n^2 grid points. In this paper we consider the problem of directly solving the class of N by N symmetric positive definite linear systems of equations

$$Ax=b, \tag{1}$$

where each x_i is associated with a grid point and A has the property that $A_{ij}\neq 0$ only if x_i and x_j are associated with nodes belonging to the same small square. We must specify how the unknowns are to be numbered if the above remark is to precisely determine the structure of A.

Our method of solution is to factor A into LL^T, where L is lower triangular, and then to solve $Ly=b$ followed by the solution of $L^Tx=y$.* The algorithm is essentially the well-known Cholesky (or square root) method, which has the agreeable property that it is numerically stable when applied to PAP^T, where P is any N by N permutation matrix.[14] Thus, we are free to permute the rows and columns of A to achieve other objectives, such as reduced computation and/or storage requirements, or convenient storage management. These objectives often compete with each other, and their relative importance depends upon the problem being solved, the characteristics of the computing system available, and programming expertise. The "best" way to number the equations depends upon which of the objectives we consider to be most important.

Certain members of our problem class which arise in connection with difference discretizations of Poisson's equation on a rectangular domain can be solved using special fast direct methods which require only $0(n^2 \log_2 n)$ arithmetic operations and $0(n^2)$ storage locations.[1,3]

For *any* system in our class, it is possible to number the equations (1) so that A can be factored in $0(n^3)$ arithmetic operations, and the number of non-zerö components in L is only $0(n^2 \log_2 n)$.[5] Unfortunately, these latter numbering schemes are somewhat complicated, and yield L's having their non-zero components scattered throughout the lower triangle. In order to achieve the above bounds on storage and computation, general sparse matrix techniques must be used. Their programming is relatively complicated, and their performance and efficiency is sensitive to hardware and software characteristics. See Gustavson[6] for a careful discussion of these methods.

On the other hand, if we number the equations in the natural row by row fashion, and employ a standard band linear equation solver,[8] the programming and data management are straightforward and convenient. Unfortunately, the computation and storage requirements are $0(n^4)$ and $0(n^3)$ respectively.

In this paper we describe a computational scheme for solving (1) which requires no sparse matrix techniques; only dense or band linear systems must be solved. Furthermore, we show that the computation and storage requirements for our scheme are respectively $0(n^3\sqrt{n})$ and $0(n^2\sqrt{n})$.

Our method and results apply with little modification to more general situations where there are nodes on the sides of the small squares, and where there is more than one unknown associated with each node.[4,15] Our scheme also applies to matrix problems which arise in connection with the use of spline bases to solve elliptic boundary value problems.[12] In this case unknowns associated with nonadjacent squares may be connected. We discuss these generalizations in our concluding remarks, but since the extensions are straightforward, for clarity

* For sparse matrix calculations, the $\bar{L}D\bar{L}^T$ factorization may be more efficient, where $\bar{L}$ is unit lower triangular and D is a positive diagonal matrix. The scheme we propose works equally well for either factorization.

we present the simplest case. We shall not, therefore, distinguish between "node" and "unknown" in the sequel.

DESCRIPTION OF THE COMPUTATIONAL SCHEME

For some positive integer $\alpha \ll n$, choose $\alpha - 1$ horizontal lines of our n by n grid which divide the mesh into α approximately equal parts, each part containing approximately n^2/α nodes. These sets of nodes (unknowns) are independent in the sense that if x_i and x_j lie in different sets, then $A_{ij} = 0$.

Consider Figure 1 below, where for definiteness we choose $\alpha = 3$.

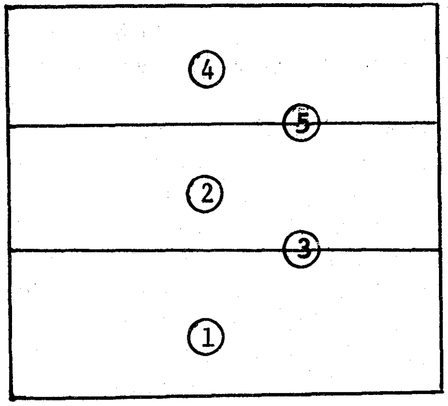


Figure 1—The n by n mesh divided into 3 parts. Circled numbers indicate the order in which node sets are to be numbered.

The lines ③ and ⑤ each consist of n nodes, and following Rose,[10] we refer to them as *separators*. The node sets designated by ①, ② and ④ will be referred to as the *independent blocks*. In general, we have α independent blocks separated by $\alpha - 1$ separators.

We number the unknowns in ① followed by those in ② *column by column*, followed by the unknowns in ③ in any order. We then number the unknowns in ④ column by column followed finally by those in ⑤ in any order.

In block form, the coefficient matrix A has the structure below

$$A = \begin{pmatrix} A_1 & & C_1^T & & \\ & A_2 & B_2^T & & C_2^T \\ C_1 & B_2 & A_3 & & \\ & & & A_4 & B_4^T \\ & C_2 & & B_4 & A_5 \end{pmatrix}.$$

Factoring A into LL^T, we obtain

$$L = \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ C_1L_1^{-T} & B_2L_2^{-T} & L_3 & & \\ & & & L_4 & \\ & C_2L_2^{-T} & F_3 & B_4L_4^{-T} & L_5 \end{pmatrix}$$

where

$$F_3^T = -L_3^{-1}B_2A_2^{-1}C_2^T,$$

$$A_i = L_iL_i^T,\ i = 1, 2, 4,$$

$$\hat{A}_3 = A_3 - B_2A_2^{-1}B_2^T - C_1A_1^{-1}C_1^T = L_3L_3^T,$$

and

$$\hat{A}_5 = A_5 - B_4A_4^{-1}B_4^T - F_3F_3^T - C_2A_2^{-1}C_2^T = L_5L_5^T. \quad (2)$$

Note that for $\alpha > 3$, the structure of the last two columns and rows of A and L would simply repeat.

Using the important observation of Rose and Bunch[11] that the matrices $B_iL_i^{-T}$ and $C_iL_i^{-T}$ will be fuller than B_i or C_i, we propose only to store the L_i's and F_i's. Having them available, the solution of our example would proceed as follows, where x and y are partitioned corresponding to A. Parentheses indicate the order in which computations are performed. Note that only triangular systems of equations are solved. For example, in 1(c) below the vector $L_4^{-T}y_4$ is obtained by solving a triangular system.

1. (a) Solve $L_iy_i = b_i$, $i = 1, 2, 4$. These can be solved in any order, or simultaneously.
   (b) Compute $b_3' = b_3 - B_2(L_2^{-T}y_2) - C_1(L_1^{-T}y_1)$ and then solve $L_3y_3 = b_3'$.
   (c) Compute $b_5' = b_5 - B_4(L_4^{-T}y_4) - F_3y_3 - C_2(L_2^{-T}y_2)$ and then solve $L_5y_5 = b_5'$.
2. (a) Solve $L_5^Tx_5 = y_5$.
   (b) Compute $y_4' = y_4 - L_4^{-1}(B_4^Tx_5)$ and solve $L_4^Tx_4 = y_4'$.
   (c) Compute $y_3' = y_3 - F_3^Tx_5$ and solve $L_3^Tx_3 = y_3'$.
   (d) Compute $y_2' = y_2 - L_2^{-1}(C_2^Tx_5) - L_2^{-1}(B_2^Tx_3)$ and solve $L_2^Tx_2 = y_2'$.
   (e) Compute $y_1' = y_1 - L_1^{-1}(C_1^Tx_3)$ and solve $L_1^Tx_1 = y_1'$.

The scheme for $\alpha > 3$ is obvious.

Before proceeding to the next section, the reader should verify that our storage needs in the example above are only $n^3/3 + 0(n^2)$, rather than the $n^3 + 0(n^2)$ required for the usual row by row numbering scheme.

We assume throughout that the so-called diagonal storage scheme[8] is used to store the band matrices.

Storage requirements and operation counts

Following Cuthill and McKee,[2] we define the bandwidth m of a symmetric matrix W by

$$m = \max_{W_{ij} \neq 0} | i-j |$$

Using the notation $\approx$ to mean "approximately," and assuming $\alpha \ll n$, we observe that the dimension of the A_i's and L_i's corresponding to the independent blocks will be $\approx n^2/\alpha$. The A's corresponding to the separators are of dimension n, and although they are sparse, their corresponding L_i's are in general full lower triangular matrices. The F_i's are in general full matrices. Recalling that there are α independent blocks, $\alpha-1$ separators, and $\alpha-2$ F_i's, and adding several more n^2 words of storage for x, b and temporary space, we obtain the following estimate $S(\alpha)$ of our storage requirements:

$$S(\alpha) = n^3/\alpha + 3\alpha n^2/2.$$

We ignore the storage required for the B_i and C_i, since their requirements are only $0(\alpha n)$.

The above is minimized when $\alpha = \hat{\alpha} = \sqrt{2n/3}$, and

$$S(\hat{\alpha}) = \sqrt{6} n^{5/2}.$$

Table 1 demonstrates the rather significant reduction in storage requirements over the usual row by row ordering scheme.

TABLE 1—Storage Requirements for our Block Scheme Compared to the Row by Row Ordering

n	N	$\sqrt{6}\, n^{5/2}$	n^3
10	100	775	1,000
20	400	4,380	8,000
30	900	12,075	27,000
40	1,600	24,800	64,000
50	2,500	43,250	125,000
100	10,000	244,950	1,000,000

We now obtain a crude estimate for the number of multiplicative operations required for our scheme. We use $\frac{1}{2}m^2N$ as an estimate for the cost of factoring an N by N symmetric positive definite matrix having bandwidth m.

First observe that the calculation of the L_i's corresponding to the independent blocks requires about $\alpha(\frac{1}{2})(n/\alpha)^2(n^2/\alpha) = (n^4/\alpha^2)/2$ multiplicative operations. The calculation of the L_i's corresponding to the separators (once we have computed the $\hat{A}_i$'s) requires approximately $(\alpha-2)n^3/6 \approx \alpha n^3/6$ operations.

Now consider the calculation of the $\hat{A}_i$'s from formulas of the type (2). An example is

$$\hat{A}_i = A_i - B_{i-1}A_{i-1}^{-1}B_{i-1}^T - F_{i-2}F_{i-2}^T - C_{i-3}A_{i-3}^{-1}C_{i-3}^T.$$

Normally, we would compute $B_{i-1}A_{i-1}^{-1}B_{i-1}^T$ by first calculating $W = L_{i-1}^{-1}B_{i-1}^T$ and then computing WW^T. However, using the structure of L_{i-1} and assuming exact numerical cancellation does not occur,** it is easy to show that W is about half full. We require at least $n^4/(3\alpha)$ multiplications to compute WW^T, and about $n^3/(2\alpha)$ auxiliary storage locations are required for W.

However, if we instead compute $\hat{W} = A_{i-1}^{-1}B_{i-1}^T$ and then utilize the fact that B has fewer than $3n$ nonzero components when computing $B_{i-1}\hat{W}$, we perform only about $2n^4/\alpha^2$ operations. Furthermore, we can compute $B_{i-1}A_{i-1}^{-1}B_{i-1}^T$ column by column, and only n^2/α temporary storage locations are required.

Using this crucial observation, the calculation of the $\hat{A}_i$'s requires about $(\alpha-1)(4n^4/\alpha^2 + n^3) \approx 4n^4/\alpha + \alpha n^3$ multiplications. Utilizing the sparsity of the B_i's and C_i's in the same way, the calculation of the F_i's requires about $(\alpha-2)(2n^4/\alpha^2 + n^3/2) \approx 2n^4/\alpha + \alpha n^3/2$ operations.

Collecting terms, we obtain the estimate $M(\alpha)$ for the number of multiplications required to produce L:

$$M(\alpha) = 6n^4/\alpha + 5\alpha n^3/3 + n^4/(2\alpha^2).$$

Assuming $\alpha \gg 1$, $M(\alpha)$ is approximately minimized for $\alpha = \tilde{\alpha} = \sqrt{18n/5}$, yielding

$$M(\tilde{\alpha}) = 2\sqrt{10}\ n^3\sqrt{n} + 5n^3/36.$$

Thus, $M(\tilde{\alpha}) > n^4/2$ unless N is very large indeed ($\approx 25{,}000$), although $M(\tilde{\alpha}) < n^4$ if N is larger than about 1,600. Thus, unless N is very large we will pay a modest premium in arithmetic operations if we use our block scheme. In exchange we obtain a substantial decrease in storage requirements. It is interesting to note that if we do not make use of our observation in computing $\hat{A}_i$'s, the operation count is $0(n^4)$.

CONCLUDING REMARKS

1. The procedure represents a considerable improvement over the standard row by row scheme; whether its comparative simplicity renders it competitive or superior to the use of more sophisticated orderings depend upon our particular computing environment and programming expertise. An attempt to at least partially answer this question is a topic of further research.

** This is a reasonable assumption in the presence of rounding error.

2. As we stated in the introduction, similar results hold for more general grid problems. When edge or interior nodes occur, one should use "profile" or "envelope" methods[4,7,9] rather than band schemes for best results. Problems arising through the use of splines have the property that unknowns associated with grid points p and q are connected provided the maximum difference in their x or y grid coordinates is bounded by some number d, which depends upon the degree of the spline. To apply our scheme we simply choose sets of d adjacent parallel grid lines as separators and proceed as before.
3. For matrix problems arising from the 3-dimensional unit cube grid having n^3 nodes, we can choose separators consisting of planes of grid points and again apply the same techniques. The computation and storage estimates achieved are respectively $0(n^6\sqrt{n})$ and $0(n^4\sqrt{n})$, compared to $0(n^7)$ and $0(n^5)$ for the standard plane-by-plane numbering scheme.
4. It seems fairly obvious that similar ideas can be applied to less regular problems, but it is difficult to obtain quantitative estimates of how much might be gained. Intuitively, we want to choose small separators, yielding independent blocks of nodes which can be numbered so as to have a small band or profile. The study of automatic schemes for doing this is another topic for future research.

ACKNOWLEDGMENT

Part of this research was performed while the author was visiting the IBM Watson Research Center, Yorktown Heights, New York. Several helpful conversations with Dr. Fred Gustavson of the research center are gratefully acknowledged. This work was supported in part by Canadian National Research Grant A8111.

REFERENCES

1 B L BUZBEE G H GOLUB C W NIELSON
On direct methods for solving Poisson's equations
SIAM Journal for Numerical Analysis 1 1970 pp 627-656

2 E CUTHILL J McKEE
Reducing the bandwidth of sparse symmetric matrices
Proc 24th National Conference Association Computer Machinery ACM Publication P 69 1122 Avenue of the Americas New York New York 1969

3 F W DORR
The direct solution of the discrete Poisson equation on a rectangle
SIAM Review 12 1970 pp 248-263

4 J A GEORGE
Computer implementation of the finite element method
Stanford Computer Science Department Technical Report STAN-CS-71-208 Stanford California 1971

5 J A GEORGE
Nested dissection of a regular finite element mesh
To appear in SIAM Journal for Numerical Analysis

6 F G GUSTAVSON
Some basic techniques for solving sparse systems of linear equations in *Sparse Matrices and Their Applications*
(D J Rose and R A Willoughby Editors) Plenum Press 1972

7 A JENNINGS
A compact storage scheme for the solution of symmetric linear simultaneous equations
Computer Journal 9 (1966) pp 281-285

8 R S MARTIN J H WILKINSON
Symmetric decomposition of positive definite band matrices
Handbook series Linear Algebra Numerische Mathematik 7 (1965) pp 355-361

9 R J MELOSH R M BAMFORD
Efficient solution of load-deflection equations
Journal of the American Society of Civil Engineers Structural Division 95 (proc paper #6510) pp 661-676

10 D J ROSE
A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations
In Graph Theory and Computing (R C Read Editor) Academic Press New York 1972

11 D J ROSE J R BUNCH
The role of partitioning in the numerical solution of sparse systems
In *Sparse Matrices and Their Applications* (D J Rose and R A Willoughby Editors) Plenum Press New York 1972

12 M H SCHULTZ
Elliptic spline functions and the Rayleigh-Ritz-Galerkin
Mathematics of Computation 24 1970 pp 65-80

13 K L STEWART J BATY
Dissection of structures
Journal of the American Society of Civil Engineers Structural Division (Proc Paper #6502) 93 pp 217-232

14 J H WILKINSON
The algebraic eigenvalue problem
Clarendon Press Oxford England 1965

15 O C ZIENKIEWICZ
The finite element method in engineering science
McGraw-Hill London 1971

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES, INC. (AFIPS)

AFIPS OFFICERS and BOARD OF DIRECTORS

President

Mr. Walter L. Anderson
General Kinetics, Inc.
12300 Parklawn Drive
Rockville, Maryland 20852

Vice President

D. Robert A. Kudlich
Raytheon Co., Equipment Division
Wayland Laboratory
Boston Post Road
Wayland, Massachusetts 01778

Secretary

Mr. Richard B. Blue, Sr.
TRW Systems Group
Scientific Data Processing Lab.
One Space Park—R3/1098
Redondo Beach, California 90278

Treasurer

Mr. George Glaser
McKinsey and Company, Inc.
3000 Sand Hill Road
Menlo Park, California 94025

Executive Director

Dr. Bruce Gilchrist
AFIPS
210 Summit Avenue
Montvale, New Jersey 07645

ACM Directors

Dr. Anthony Ralston
SUNY at Buffalo
Computer Science Department
4226 Ridge Lea Road
Amherst, New York 14226

Mr. Donn B. Parker
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, California 94025

Mr. Herbert S. Bright
Computation Planning, Inc.
5401 Westbard Avenue, Suite 520
Washington, D.C.

IEEE Directors

Dr. A. S. Hoagland
IBM Corporation
Dept. 29A—Building 021
P.O. Box 1900
Boulder, Colorado 80302

Professor Edward J. McCluskey
Stanford University
Department of Electrical Engineering
Palo Alto, California 94305

Dr. S. S. Yau
Department of Electrical Engineering
Stanford University
Palo Alto, California 94305

Simulations Council Director

Mr. Frank C. Rieman
Electronic Associates, Inc.
P.O. Box 7242
Hampton, Virginia 23366

Association for Computation Linguistics Director

Dr. A. Hood Roberts
Center for Applied Linguistics
1717 Massachusetts Avenue, N.W.
Washington, D.C. 20036

American Institute of Aeronautics and Astronautics Director

Mr. Frank Riley, Jr.
Auerbach Corporation
1501 Wilson Boulevard
Arlington, Virginia 22209

American Statistical Association Director

Dr. Mervin E. Muller
5303 Mohican Road
Mohican Hills
Washington, D.C. 20016

Instrument Society of America Director

Mr. Theodore J. Williams
Purdue Laboratory for Applied Industrial Control
Purdue University
Lafayette, Indiana 47907

Society for Information Display Director

Mr. William Bethke
Rome Air Development Center
RADC (IS, W. Bethke)
Griffiss AFB, New York 13440

American Institute of Certified Public Accounts Director

Mr. Noel Zakin
AICPA
666 Fifth Avenue
New York, New York 10019

American Society for Information Science Director

Mr. Herbert Koller
ASIS
1140 Connecticut Avenue, N.W., Suite 804
Washington, D.C. 20036

Society for Industrial and Applied Mathematics Director

Dr. D. L. Thomsen
IBM Corporation
Armonk, New York 10504

Special Libraries Association Director

Mr. Herbert S. White
Institute for Scientific Information
325 Chestnut Street
Philadelphia, Pennsylvania 19105

Association for Educational Data Systems Director

Dr. Sylvia Charp
Director of Instructional Systems
The School District of Philadelphia
Board of Education
5th and Luzerne Streets
Philadelphia, Pennsylvania

JOINT COMPUTER CONFERENCE BOARD

President

Mr. Walter L. Anderson
General Kinetics, Incorporated
12300 Parklawn Drive
Rockville, Maryland 20852

Vice President

Dr. Robert A. Kudlich
Raytheon Company, Equipment Division
Wayland Laboratory
Boston Post Road
Wayland, Massachusetts 01778

Treasurer

Mr. George Glaser
McKinsey and Company, Iuc.
3000 Sand Hill Road
Menlo Park, California 94025

ACM Representative

Dr. Herbert R. J. Grosch
National Bureau of Standards
Center for Computer Science
Washington, D.C. 20234

IEEE Representative

Dr. S. S. Yau
Department of Electrical Engineering
The Technological Institute
Northwestern University
Evanston, Illinois 60201

SCI Representative

Mr. Paul W. Berthiaume
Electronic Associates, Inc.
185 Monmouth Park Highway
West Long Branch, New Jersey 07764

JOINT COMPUTER CONFERENCE COMMITTEE

Mr. Jerry L. Koory, Chairman
H-W Systems
525 South Virgil
Los Angeles, California 90005

JOINT COMPUTER CONFERENCE TECHNICAL PROGRAM COMMITTEE

Mr. Henry S. MacDonald, Chairman
Bell Laboratories
Murray Hill, New Jersey 07971

1973 NATIONAL COMPUTER CONFERENCE CHAIRMAN

Dr. Harvey Garner
Director
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania 17104

1972 FJCC STEERING COMMITTEE

Chairman
Robert Spinrad
Xerox Corporation

Technical Program
Donald A. Meier
National Cash Register

Secretary
Harold Sarkissian
Major Data Corp.

Controller
Howard Verne
Hughes Aircraft Co.

Registration
Patricia Riley
TRW Systems

Local Arrangements
Antonia Schuman
Litton Industries

Printing and Mailing
Katherine Jamerson
Computer Sciences Corp.

Exhibits
A. Luke Ward
San/Bar Electronics Corp.

Public Relations
Allen T. LeAnce
LeAnce and Associates

Special Activities
Fred Gruenberger
San Fernando Valley State College

TECHNICAL PROGRAM COMMITTEE

Chairman
Mr. Donal A. Meier
National Cash Register

Vice-Chairman
Dr. Harold Petersen
RAND Corporation

Speaker Arrangements Director
Mr. Lynn Maxson
IBM Corporation

Liaison & Review Coordinator
Mr. Wolfgang G. Pfeiffer
National Cash Register

Publication Director
Mr. Russell Bennett
Burroughs Corporation

SESSION DIRECTORS

Analysis and Simulation Director
Dr. Ray Nilsen
University of California, Los Angeles

Interdisciplinary Director
Mr. Lowell Amdahl
Compata, Inc.

Software Director
Dr. Richard R. Muntz
University of California, Los Angeles

Users and Applications Director
Mr. Ross F. Penne
University of Southern California

Users and Applications Assoc. Dir.
Dr. Arnold F. Goodman
McDonnell-Douglas Astronautics

Hardware Director
Mr. Jack Pariser
Hughes Aircraft Co.

Systems and Architecture Director
Mr. Harut Barsamian
National Cash Register

SESSION CHAIRMEN, REVIEWERS AND PANELISTS

SESSION CHAIRMEN

Baker, Frank
Balzer, Robert M.
Barsamian, Harut
Bekey, George
Boehm, Barry
Chen, T. C.
Chu, Wesley W.
Denning, Peter J.
Farber, David J.
Fetter, William A.
Flynn, Michael J.
Gaines, Eugene C., Jr.
Gentile, Richard B.
Golub, Eugene
Goodman, Arnold

Hamming, Richard W.
Hollander, G.
Hunter, Kenneth
Husson, Samir
Juncosa, M. L.
Kimbleton, Stephen
Kiviat, Philip J.
Lyon, John K.
McCluskey, E. J.
McManus, Jack
McNamee, Laurence
Mason, Maughn
Mills, Harlan
Mitchell, Gordon

Montgomery, Christine
Morgan, Howard
Newport, Christopher
Patrick, Robert
Penne, Ross F.
Phister, Montgomery, Jr.
Pinkerton, Tad
Reinstedt, Robert
Stefferud, Einar
Taplin, Janet M.
Turn, Rein
Waxman, Ronald
Weissman, Clark
Wilson, Jon C.

REVIEWERS

Alberts, A.
Alrich, J. C.
Anderson, H. M.
Anderson, R.
Arndt, F.
Arnovick, G.
Astrahan, M.
Augustin, D. C.
Avizienis, A.
Ball, N.
Barlow, A. E.
Becker, P.
Bell, T. E.
Bernstein, W. A.
Biener, J. W.
Bloomfield, J.
Boehm, B. W.
Borgsahl, R.
Bork, A.
Branch, R.
Branin, F.
Brereton, T. B.
Brown, A. B.
Calhoon, D.
Canova, G.
Cardwell, D.
Carlson, G.
Carroll, J.
Carter, W. C.
Chen, T. C.
Chernak, J.
Cheydler, B. F.
Choma, J. Jr.
Chu, W. W.

Climenson, W. D.
Copp, D. H.
Courtney, R.
Cowell, W.
Critchlow, A.
Csuri, C.
Dale, A.
Dalrymple, S. H.
Darms, D.
Dittberner, D.
Dorr, F. W.
Duggan, M.
Dumey, A. I.
Eccles, W.
Edwin, L.
Eisenstark, R.
Farmer, N. A.
Feurzeig, W.
Feustel, E. A.
Fiefant, R.
Firschein, O.
Fletcher, J.
Frank, H.
Freilich, A.
Frost, C. R.
Fuches, E.
Fulton, L. M.
Gardner, R.
Gentile, R.
Gillette, G.
Gilliland, B.
Gold, M.
Goodman, A. F.
Gosden, J.

Gotterer, M.
Grandmaison, J.
Grau, A.
Grobstein, D.
Gulick, L. R. Jr.
Hagenstad, M. T.
Hamilton, D. C.
Hammer, C.
Hamming, R. W.
Hammond, F.
Hanson, R. J.
Harper, S.
Harrison, R. L.
Hartwick, R.
Hendrie, G.
Herr, W. B.
Heterick, R. Jr.
Hixon, J.
Hoffman, L.
Hootman, J. T.
Humphrey, R.
Hunt, E.
Hunter, K. W.
Hutt, A. E.
Hyman, M.
Isaksen, L.
Ito, R. A.
Jackson, H. L.
Jeffrey, S.
Jellinek, I.
Jenkins, J. M.
Joseph, E.
Kaltman, A.
Karplus, W. J.

Kay, A.
Keenan, T.
Kernighan, B. W.
Kerr, D. V.
Kimbleton, S. R.
Klein, E.
Kleinrock, L.
Klinger, A.
Klotz, D. A.
Knight, K.
Koory, J.
Kosinski, W.
Kurasch, C.
Kuhns, J. L.
Lange, L.
Larkin, R.
Larson, K.
Lasser, D. J.
Ledley, R.
Leffler, N.
Leichner, G. H.
Levine, L.
Lewis, W.
Lindloon, E.
Linville
Liskov, B.
Loewe, R. T.
Logan, R. S.
Losleben, P.
Luderer, G. W. R.
Lum, V.
Madden, J.
Markel, R.
Marks, H.
Martin, W.
Mathison, S.
Mathur, F.
Mayper, V.
McCracken, D.
McGovern, W.
McIssac, D.
McMurran, M. N.
Meier, D.
Mekota, J.
Mergenweck,
Meuller, M.
Michle, M.
Miller, S.
Miller, W. G.
Mills, H.
Minker, J.
Mitchell
Mittman, B.
Moler, C. B.
Morterana
Myers, R.
Nance, R. E.
Nicols, A. J.
Niedrauer, R. V.
Nielsen, R.
O'Brien, J.
Ofek, H.
Oliver, P.
Onovec
Onyshkevych
Opderbeck, H.
Ostapko, D.
Owens, J.
Pariser, J.
Parker, D.
Patel, A.
Patrick, R. L.
Penne, R.
Petersen, H.
Phillips, T. D.
Pohm, A.
Pomerene, S.
Postel, J.
Price
Prokop, J.
Rajaraman, A. S.
Ramamoorthy, C.
Ray, L.
Reynolds, C.
Rhodes
Rick, J. W.
Rigney, J.
Ripley, G.
Robinson, J.
Robinson, L.
Rodriguez, R.
Rosenbaum, S.
Rosenberg, A. M.
Rosenthal, M.
Rutman, R.
Saal, H.
St. John, D.
Schafer, E.
Schell, R.
Schichman, H.
Schieldge, J.
Schischa, E.
Schneidewind, N.
Schultz, M. H.
Sedelow, W.
Schechter, J.
Short, G. E.
Silvern, L.
Singh, S.
Skelly, P. G.
Small, D. L.
Smith, C.
Smith, R. A.
Southworth, R. W.
Steenbergen, H.
Stefferud, E.
Stephenson, J. W.
Stewart, R. M.
Sturm, W.
Su, S. Y. H.
Summit, R.
Sutherland, W.
Svoboda, A.
Sykes, D.
Taylor, R.
Thomas, R. T.
Tseng, C.
Tucker, S.
Uhlig, R. H.
Uttal, W.
Van Tassel, D.
Walker, D. E.
Watson, R. A.
Watt, W. C.
Weeg, G. P.
Wegbreit, B.
Weiss, E.
Weissman, C.
Werner, J. J. Jr.
Wersan, S.
Whitney, D.
Wiederhold, G.
Wigington, R.
Wiggins
Wilkov, R. S.
Williams, J. G.
Williams, L.
Williams, T. J.
Wilner, W.
Wilson, J.
Wolf, E. W.
Wright, K.
Wyllys, R.
Yakowitz, S.
Yelvington, S.
Young, J.
Zelkovitz, M.

PANELISTS AND SPEAKERS

Donald Aufenkamp, N.S.F.
A. Avizienis, University of Southern California
John Bacon, United California Bank
Max Beere, Tymshare
Barry Boehm, RAND Corporation
Robert Brass, Xerox
Barry Brotman, Allied Chemical Corporation
Gary Carlson, Brigham Young University
Leo Cohen, Consultant
David Copp, Bell Telephone Laboratories
Stephen Crocker, Department of Defense
John Davis, TESDATA Systems Corp.
Lt. Col. Phillip Enslow Jr., Office of Telecommunications Policy, Executive Office of the President
David Evans, Evans and Sutherland
John Farquhar, RAND Corporation
Nick Finamore, Western Electric
H. Fleisher, IBM Corporation
L. Garrett, Motorola
Robert Gordon, Consultant
P. F. Gudenschwager, Honeywell
Richard Hamming, Bell Telephone Laboratories
Cdr. Grace Murray Hopper USNR
Richard Johnson, Stanford University Computation Center
Robert Johnson, Burroughs Corporation
V. Kahan, University of California at Berkeley
Robert Kahn, Bolt, Beranek and Newman, Inc.
E. Mahoney, United States General Accounting Office
C. H. Mays, Fairchild
John McCarthy, Stanford University
M. Douglas McIlroy, Bell Telephone Laboratories
Harry Mergler, Case Western Reserve University
Capt. M. Morris, Federal ADP Simulation Center
Mervin Muller, International Bank for Reconstruction and Development
Peter Newcombe, Brigham Young University
Nils Nilsson, Stanford Research Institute
A. Patel, IBM Corporation
Alan Perlis, Yale University
Charles Perry, McDonnel-Douglas Astronautics
Tom Poole, United Computer Systems
C. Ramamoorthy, University of Texas
Louis Robinson, IBM Corporation
Arthur Rosenberg, Informatics
Capt. Paul Roth, Fleet Combat Direction Systems Support Activity
Stephen Y. Su, University of Southern California
Lee Talbert, Packet Communications, Inc.
L. C. Thomas, Bell Telephone Laboratories
D. E. Walker, S.R.I.
P. Weber, Lane County
Mark Wells, Los Alamos Scientific Laboratory
James Williams, United States General Accounting Office
Joe Wineke, Compress, Inc.
M. Worthy, Operating Systems
Gordon Zeller, *Los Angeles Times*

PRELIMINARY LIST OF EXHIBITORS

Addison-Wesley Publishing Company, Inc.
Addmaster Corporation
Addressograph Multigraph Corporation
AFIPS Press
American Elsevier Publishing Company
American Telephone & Telegraph
Ampex Corporation
Anaheim Publishing Company
Ansul Company
Basic Timesharing, Inc.
Beehive Terminal
Bridge Data Products, Inc.
Burroughs Corporation
Caelus Memories, Inc.
Centronics
Century Electronics and Instruments
Cipher Data Products
Codex Corporation
Collins Radio Company
ComData Corporation
Computer Access Systems, Inc.
Computer Automation, Inc.
Computer Copies Corporation
Computer Design Publishing Corporation
Computer Machinery Corporation
Controls Research Corporation
Courier Terminal Systems, Inc.
Data Disc, Inc.
Data General Corporation
Datamation
Data Printer Corporation
Datapro Research Corporation
Data Products Corporation
Dataram Corporation
Datawest Corporation
Datum, Inc.
Diablo Systems, Inc.
Digital Computer Controls, Inc.
Digital Development Corporation
Documation, Inc.
DuPont Company
Eastman Kodak Company
Electronic Engineering Company of California
Electronic News, Fairchild Publications
Facit-Odhner, Inc.
Federal Screw Works
Floating Point Systems, Inc.
General Automation, Inc.
GTE Lenkurt
G-V Controls
Hayden Publishing Company, Inc.
Hewlett-Packard Company
Honeywell Computer Journal
Houston Instrument
IMSL
Inforex, Inc.
Information Data Systems, Inc.
Infosystems
Infoton, Inc.
Intel Corporation
International Communications Corporation
International Computer Products, Inc.
Kennedy Company
Kybe Corporation
Lipps, Inc.
Litton ABS OEM Products
Lorain Products Corporation
Marubeni America Corporation
Microdata Corporation
Micro Switch
Milgo Electronic Corporation
Miratel Division—Ball Brothers Research Corp.
Modern Data
Mohawk Data Sciences Corporation
Northern Electric Company, Ltd.
Nortronics Company, Inc.
Olympia USA, Inc.
Ovonic Memories, Inc.
Panasonic
Paradyne Corporation
Pertec Corporation
Pioneer Electronics Corporation
Pioneer Magnetics, Inc.
Potter Instrument Company, Inc.
Prentice Hall, Inc.
Printer Technology, Inc.
Producers Service Corporation
Radley Associates Limited
Randomex, Inc.
Raymond Engineering, Inc.
Raytheon Service Company
Redactron Corporation

Remex, A unit of Ex-Cell-O Corporation
Sangama Electric Company
Signal Galaxies, Inc.
The Singer Company
Sycor, Inc.
Sykes
Systems Furniture Company
Tally Corporation
Techtran Industries, Inc.
Tekronix, Inc.
Tele-Dynamics
Teleprocessing Industries, Inc.
Teletype Corporation
Texas Instruments, Inc.
Toko, Inc.
Tri-Data Corporation
Van San Corporation
Vector General, Inc.
Velo-Bind
Wangco, Inc.
John Wiley and Sons, Inc.
Xerox Corporation

AUTHOR INDEX