LAMS-2607

C.3 [redacted]

LOS ALAMOS SCIENTIFIC LABORATORY
OF THE UNIVERSITY OF CALIFORNIA ○ LOS ALAMOS NEW MEXICO

THE IVY SYSTEM

Printed in USA Price $ 3.50. Available from the

Office of Technical Services
U. S. Department of Commerce
Washington 25, D. C.

LAMS-2607
MATHEMATICS AND COMPUTERS
(TID-4500, 16th Ed.)

LOS ALAMOS SCIENTIFIC LABORATORY
OF THE UNIVERSITY OF CALIFORNIA LOS ALAMOS NEW MEXICO

REPORT WRITTEN: August 1961

REPORT DISTRIBUTED: October 6, 1961

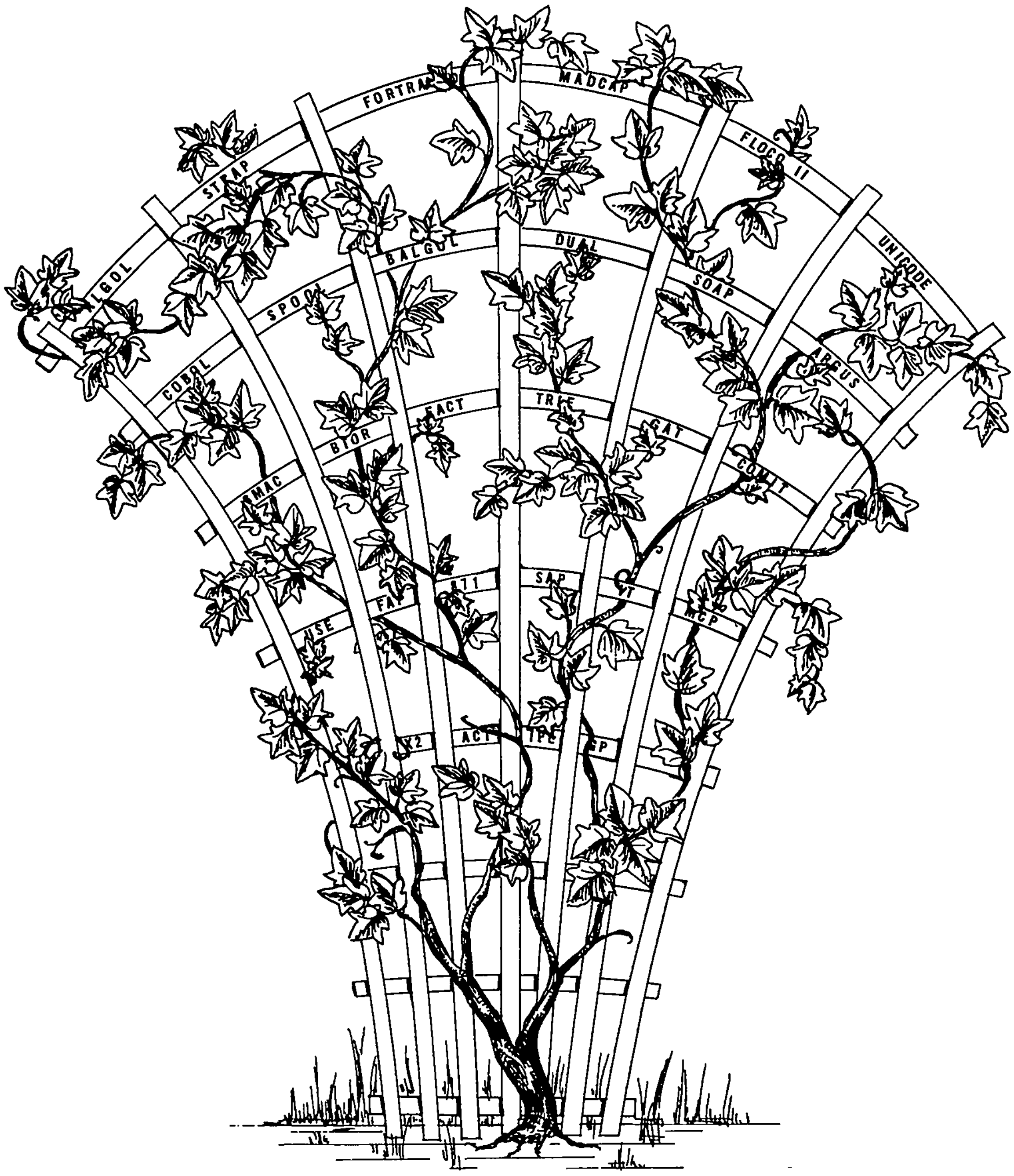
THE IVY SYSTEM

by

Forrest W. Brinkley
Bengt G. Carlson
Chester S. Kazek, Jr.
Clarence E. Lee
Zane C. Motteler

MANUAL EDITOR: Zane C. Motteler

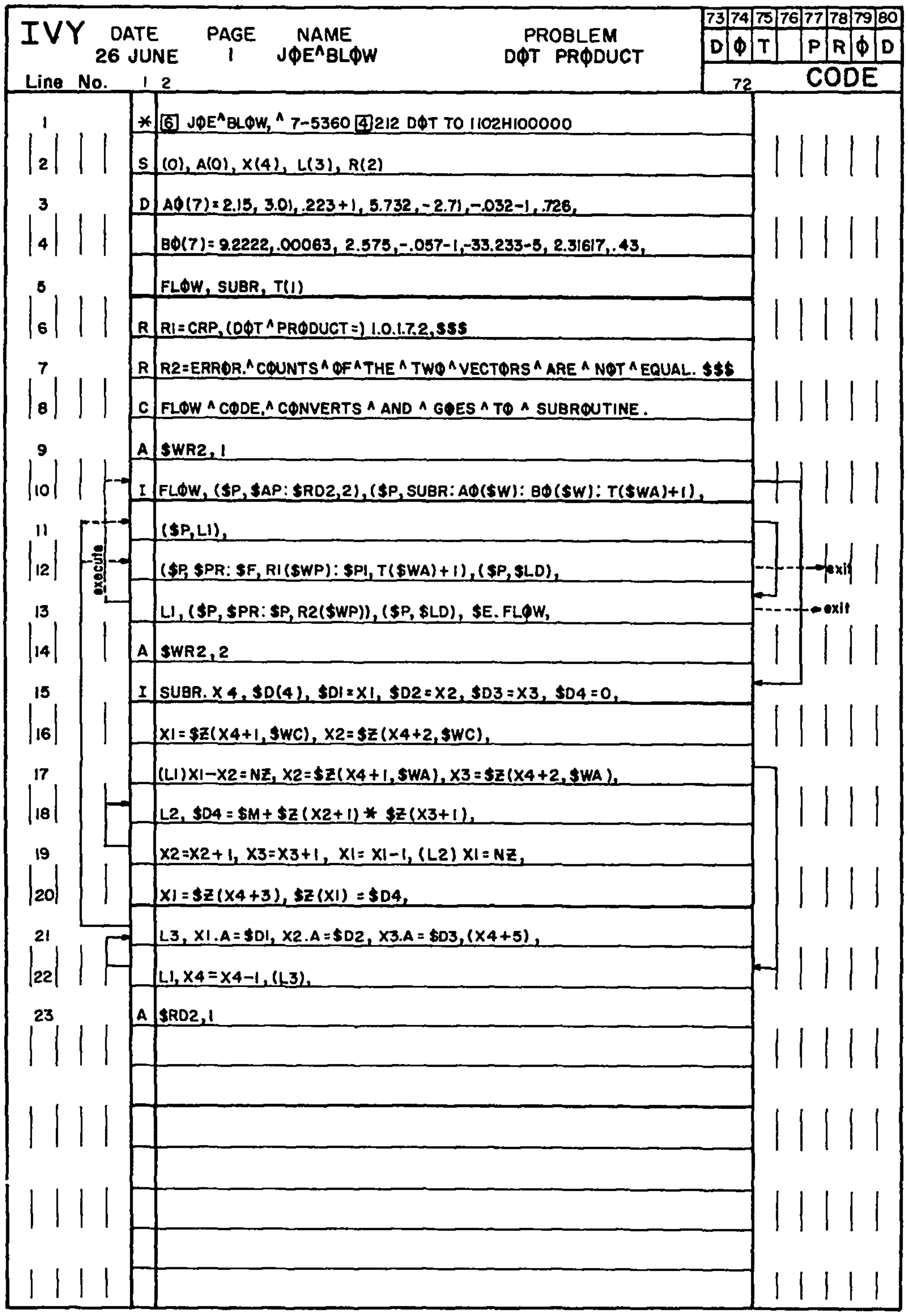
Contract W-7405-ENG. 36 with the U. S. Atomic Energy Commission


FORTRA
MADCAP
STRAP
FLOCO II
LGOL
UNICODE
BALGOL
DUAL
SPOO
SOAP
COBOL
ARGUS
FACT
TRIE
BIOR
GAT
MAC
COMIT
711
SAP
FAP
IT
USE
MCP
X2
ACT
IPL
EP

IVY

ABSTRACT

IVY, an algebraic coding system for the IBM 7090 and 7030 electronic data processing machines, is described. A sample code is first illustrated for purposes of familiarization. The general features of the IVY system are then discussed in the Introduction. The body of the text discusses card types, the entry of data, remarks, and calling sequences, and the formats for writing code in the IVY algebraic language. Finally, subroutines incorporated in the IVY system and error indications given by the system are described, and some coding examples are shown. The final chapter is composed of tables for reference purposes. The appendices discuss more sophisticated coding techniques and the longhand coding conventions for the 7090 and 7030.



```
IVY  DATE       PAGE   NAME                PROBLEM               73 74 75 76 77 78 79 80
     26 JUNE     1     JØE^BLØW            DØT PRØDUCT           D  Ø  T     P  R  Ø  D
Line No.        1 2                                              72          CODE

 1   *  [6] JØE^BLØW,^ 7-5360 [4]212 DØT TO 1102H100000
 2   S  (0), A(0), X(4), L(3), R(2)
 3   D  AØ(7)=2.15, 3.01, .223+1, 5.732, -2.71, -.032-1, .726,
 4      BØ(7)= 9.2222, .00063, 2.575, -.057-1, -33.233-5, 2.31617, .43,
 5      FLØW, SUBR, T(1)
 6   R  R1=CRP, (DØT^PRØDUCT=) 1.0.1.7.2, $$$
 7   R  R2=ERRØR.^CØUNTS^ØF^THE^TWØ^VECTØRS^ARE^NØT^EQUAL. $$$
 8   C  FLØW^CØDE,^CØNVERTS^AND^GØES^TØ^SUBRØUTINE.
 9   A  $WR2, 1
10   I  FLØW, ($P, $AP: $RD2,2), ($P, SUBR: AØ($W): BØ($W): T($WA)+1),
11      ($P, L1),
12      ($P, $PR: $F, R1($WP): $P1, T($WA)+1), ($P, $LD),          exit
13      L1, ($P, $PR: $P, R2($WP)), ($P, $LD), $E. FLØW,           exit
14   A  $WR2, 2
15   I  SUBR. X4, $D(4), $D1=X1, $D2=X2, $D3=X3, $D4=0,
16      X1=$Z(X4+1, $WC), X2=$Z(X4+2, $WC),
17      (L1)X1-X2=NZ, X2=$Z(X4+1, $WA), X3=$Z(X4+2, $WA),
18      L2, $D4=$M+$Z(X2+1) * $Z(X3+1),
19      X2=X2+1, X3=X3+1, X1=X1-1, (L2) X1=NZ,
20      X1=$Z(X4+3), $Z(X1) =$D4,
21      L3, X1.A=$D1, X2.A=$D2, X3.A=$D3, (X4+5),
22      L1, X4=X4-1, (L3),
23   A  $RD2, 1
```

A SIMPLE AND COMPLETE IVY CODE

PREFACE

The facing page illustrates a complete, though trivial, code in the IVY language, for finding the dot product of two vectors. This is included at the start of the manual in order to familiarize the reader immediately with the appearance of a finished IVY code. As the discussion in the manual proceeds, the reader can occasionally refer back to this example for enlightenment on some of the techniques discussed. Finally, in Chapter 8, a discussion of the organization and philosophy of this code will occur, a discussion which applies to any IVY code regardless of its length.

ACKNOWLEDGEMENTS

The editor is indebted to Bengt Carlson, Chester Kazek, Jr., Forrest Brinkley, and Clarence Lee for their careful proof reading of this manual, much of which was done on their own time, and for their many helpful suggestions; to Justine Stehl, for producing the IVY tree; to Sue Vandervoort, for typing the rough draft; to Grace Cole, for typing the final draft; and to Bea Hindman, for her excellent job in lettering the examples and drawing the illustration.

CONTENTS

INTRODUCTION

The coding system described in this manual, the IVY system, represents a considerable extension, sophistication, and simplification of earlier attempts by the authors on the design of an efficient and practical coding system for both the casual and experienced programmer. Frequently an individual, usually called a "programmer" or "coder" in this manual, concerned with the solution of a complicated problem, must resort to the use of computers. For such people IVY was designed. Detailed knowledge of the behavior of various different computers is not required, but, if available, it can be applied when <u>very</u> fancy coding techniques (presumably by an experienced programmer conversant with a particular class of machines) are warranted. However, it is believed that the vast majority of problems in mathematical physics amenable to computer solution can be solved adequately, almost in their entirety, in the simple algebraic language supplied by IVY.

The IVY system is a load-and-go, one-pass compiler-assembler consisting of an algebraic language which can be used on any of a class of computers for which the system is designed, as well as facilities for coding in the language of the particular computer on which a program is being run. The main purpose of the system is to simplify and expedite the programming of problems and the debugging of resulting codes, the scheduling of machine time in installations with two or more types of machines, the exchange of codes, and the use of these at other installations. Another purpose of the system is to provide a load-and-go compiler which gives the programmer closer touch with the computer hardware, besides supplying numerous other new and unique features, many of which have never before been offered in any system of this type.

The IVY algebraic coding system has been designed for coders who are somewhat familiar with electronic computers and programming techniques, but who do not have a detailed knowledge of a particular computer. The algebra itself is written in a system called <u>machine</u> algebra, as opposed to FORTRAN and other algebraic coding systems which simulate <u>display</u> algebra, that is, the algebra of equations and formulas in the traditional mathematical sense. This machine algebra is a system similar to <u>display</u> algebra except in conventions regarding the use of parentheses. In addition, the coder is allowed (and often required) to specify actual index registers (unlike FORTRAN), to utilize a "store-address" feature, and to construct loops and sequences of code fully as complex as those possible in longhand coding, without the many restrictions imposed by FORTRAN-like systems. A code in the IVY algebraic language will be accepted, unchanged, by any computer for which IVY is available.

As previously mentioned, a longhand coding system is available in IVY, which allows the entry of <u>any</u> instructions in the instruction set of the particular machine being used, following IVY addressing conventions.

Of course, use of this feature will make an IVY code incompatible with machines of a different type. Nevertheless, in practice such longhand portions of a code are usually short, and a separate set of longhand cards can be produced for each computer, and one set substituted for another when one changes computers. For the programmer who is interested only in longhand coding for a particular machine, IVY presents a fast load-and-go longhand coding system.

The "IVY" system consists essentially of three parts: the loading program ($\emptyset$LD), the assembly program ($\emptyset$AP) and various subroutines (print, punch, tape manipulation, etc.). Only that portion of IVY currently in use is in core memory at any one time; a master control program calls in other packages as needed. Thus all but a few thousand words of core are available to the problem program. Core storage is never taken up by unconverted code, which, instead, is written on a tape designated by the programmer at initial loading time. Once a program is debugged, this tape may be saved and used to load the program whenever it is run thereafter, saving some machine time, since this tape contains a condensed version of the code. This tape will in general not be interchangeable among machines of different type for which IVY is available, since the condensed code on the tape is in a partially assembled form.

Each IVY deck begins with an "S" or "start" card, which initializes IVY for a new program. (IVY programs can be stacked one behind another in the card reader or on a BCD tape prepared by off-line card-to-tape equipment). This "S" card also contains specifications of the basic quantities of the particular program, such as the number of independent "store address" quantities, the number of index registers desired (which may be more than the particular machine contains, in which case the extra index registers are simulated with a slight loss of efficiency), the number of formulas desired, and the maximum number of branch references within a formula. Following the "S" card, cards controlling the definitions and loading of data, remarks, and calling sequence entries, may occur. The instruction cards are normally at the end of a deck. Preceding, and intermixed with, the instruction cards are "A" or "assemble" cards which control the writing of the code on tape and its subsequent conversion into machine language. The code may be followed by an "X" or "execute" card, which specifies the formula set at which execution starts.

The chief advantage of IVY, aside from its simplicity, is that no preliminary processing is necessary, such as obtaining binary cards from a separate assembly program. Thus, not only is the assembly process immediately under the programmer's control at all times, but also the source deck and object deck are one and the same. Corrections can be made in the source deck without the necessity of a tedious reassembly to obtain a new object deck. Furthermore, because of a unique new type of coding form, one comes closer than ever before to punching cards directly from the flow chart. And finally, IVY contains a feature which enables one to obtain a listing of his code if desired, at the loading time.

During its one-pass examination of the source deck IVY detects a great many different types of errors. If a detectable error occurs, IVY prints out the contents of the card on which the error occurred, one or more symbols to aid in localizing the error on the card, and a number. This number can be looked up in a table which is available at the console of each machine for which IVY is available, and which will be distributed to manual holders separately from this manual. The table entry gives an exact description of the error. It is in the detection and treatment of errors that one of the chief advantages of IVY occurs. If errors are detected in code, for instance, the programmer is still permitted to execute his program up to that point where the first executed error was detected. From this point a transfer is made to IVY, which prints out a comment to the effect that execution cannot proceed further, and gives some indication as to where this point is located. Similarly, if a data block has been defined or loaded incorrectly, references to this block are replaced by similar transfers. Thus the programmer, in a debugging run, obtains not only information on coding errors detectable by IVY, but also the results of executing the problem code to the point of the first error encountered in execution, allowing him to ferret out both coding errors and logical errors in one and the same run. As far as is known, IVY is the first programming system ever designed to allow this feature. Of course it is possible that errors detected may be of such a magnitude as to make compilation impossible, in which case IVY will suppress execution. However, it must be asserted that errors of such magnitude seldom occur, and that IVY is unique in failing to penalize programmers for minor programming errors, by allowing execution when possible. (No claim can be made that all possible detectable errors are caught, since to do this would require an impractically long program. Hopefully a useful balance between detectable and non-detectable errors has been maintained.)

The IVY system and its features, as outlined in this Introduction, are discussed in detail in subsequent chapters, with special emphasis on the algebraic system and its conventions. A knowledge of the algebraic addressing conventions is necessary to code in one of the particular long-hand systems, which therefore are described in appendices at the end of the manual, briefly but completely, and in a manner assuming some familiarity with earlier chapters, and, of course, the particular computers being utilized.

CHAPTER 1

PRELIMINARY REMARKS

Character set. The character set used by IVY is the well-known Hollerith set, i.e., the character set used by FORTRAN, which is available on the IBM 026 punch. This set consists of the alphabetic upper-case characters, the numbers 0-9, and a few punctuation marks and special characters. Limited as it is, this character set will be used until such time, if ever, as extended character set keypunches (IBM 9210) become generally available. For reference purposes, the Hollerith set consists of the characters 0 (numeric zero), 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, L, M, N, Ø (alphabetic "Ø"), P, Q, R, S, T, U, V, W, X, Y, Z, +, -, *, /, =, ', ., :, $, (,), comma, and blank.

Symbols. IVY symbols and symbolic names (with the exception of a few special symbols mentioned later) must consist only of alphabetic characters, that is, of the characters A, B, C,...,Z. Symbols may be of any length up to 6 characters. Examples: AXB, SAM, TEMP, C, PQRX, VELØCY.

Special symbols. Certain symbols for internal IVY subroutines, data blocks, and operation conventions, start with the character "$", which is not available as a symbol for IVY remarks, data, or code. Two of these symbols, $LD and $AP, were encountered in the Introduction. All "$" symbols will be discussed and defined as the need arises; a table of "$" symbols appears in Chapter 9, page 187. Note that _only_ these _internal_ IVY symbols begin with "$": _any_ symbol defined by the programmer _must_ begin with an alphabetic character.

In addition, the programmer may define symbols for certain _numbered_ quantities and _numbered_ blocks on the "S" card (see below, page 26), e.g., A1, A2, etc. for stored addresses (page 28); X1, X2,..., for index registers (page 28); L1, L2,..., for internal branch references (page 28); as well as numbered blocks beginning with an alphabetic symbol as defined above, used only to represent remarks and calling sequence blocks (page 29).

[*]_The Symbol Table._ All programmer-defined symbols are placed in the IVY symbol table. This table consists of two parts: (1) a twenty-six entry table, with each entry corresponding to one letter of the alphabet, which is always in core, and (2) a variable length table having one entry for each symbol of two or more characters, which is constructed by IVY as the symbols are defined. Each entry of these tables contains the following items of information: the symbol itself, in BCD; two addresses used by IVY for searching purposes; and a control word, containing a count of the items of information in the block, a flag indicating what type of information is loaded, and the base address of the block minus one. Once constructed, the symbol table is always in core, available to both IVY and the problem program. With the exercise of due caution, the problem program may consult and alter the symbol table at will, using conventions described in Appendix 1.

[*]Paragraphs marked with "*" and single-spaced, while informative, are not essential to the understanding of the IVY system, and can be skipped if desired.

The Order of Definition of Symbols. Since IVY is a one-pass program, all symbols must be defined before they may appear in the definition of another symbol and before they are referred to by code. Symbols can be defined on "S", "D", and "R" cards, described on pages 26, 43, and 61. Furthermore, all symbols must be defined before any code is converted, regardless of whether the code refers to the symbols or not. Since the symbol table is loaded in core immediately preceding converted code, the symbol table must be full to avoid destroying code with new entries.

All symbols must be defined on "S", "D", or "R" cards except for symbols consisting of a single alphabetic character, other than the special symbols "A", "X", and "L" (pages 28-29). Single-character symbols never need be defined since IVY always contains the 26-entry table for the single alphabetic characters. Note that by a symbol being defined is meant that the symbol must be entered in the table, although it need not have been assigned an address, value, or length unless the conversion of code or definition of another symbol requires such assignment. (For detailed instructions on defining and/or assigning values to symbols, see pages 43-61.)

*The Role of the Control Word in Error Detection. By examining the flag of the control word for a symbol, mentioned above, page 18, IVY detects such obvious errors as attempting to perform arithmetic on code and remark blocks and attempting to transfer to data or remark blocks from the problem program. If the entire control word is zero, meaning a symbol has been defined but the block has not been loaded, such errors as referring to the block in arithmetic instructions and

*Detailed discussions of the symbol table and control word formats, of interest only to the more-than-casual coder, will be found in Appendix 1 and in the various appendices relating to particular machines for which IVY is available.

attempting to define another symbol in terms of this one can be discovered. References to undefined symbols are, of course, easily detected because of the absence of the symbol from the symbol table. Ordinarily these errors are not of such magnitude as to inhibit compilation of the problem program, and whenever this is true, execution is allowed to proceed to that point where the code is first affected by such an error.

CHAPTER 2

CODING FORMS AND TYPES OF CARDS

Coding Forms. There are two forms available for IVY programming. The first form is divided into one column for the control punch (described later in this chapter) and 71 columns for the entry of information, with the last eight columns left for program identification. The contents of the program identification columns are not available to the program. The second form is similar to the first, and in addition it contains guide lines in the margins for drawing arrows, to mark flow of control, thus making the coding sheet, in essence, a flow chart. These arrows are not punched on the cards, but are merely intended as a convenience to aid the programmer in reading his code, and in making it easier for others to follow and understand the flow of the program. This feature should also lessen, if not eliminate, the need for flow charts.

Control Punches. The first column of IVY cards is always reserved for the control punch. The function of the control punch is to designate the type of information found on the cards, and to give instructions to the compiler, or both. A card containing a blank in column 1 is assumed to be a continuation of the previous card and to contain the same type of information. Certain types of cards may not have a continuation card following them; this is noted, when applicable, in the following description of the particular card types. The continuation of "R", "K", or "T" cards if any, must contain a blank in column 1, as explained below.

Identification Card. An "identification card" must precede any code which produces off-line output for printing, punching, or plotting on the SC-4020. For consistency this card should precede all IVY decks. (This card is the standard ID for the IBM 7090, as adopted by the Los Alamos Scientific Laboratory and described in a bulletin distributed to 7090 users dated April 14, 1961.) The function of this card is to identify any off-line output (listings, cards, microfilm, etc.) with the programmer's name and telephone number, so that it can easily be separated from other programmers' output and delivered to the individual concerned. To aid the operator in logging, the contents of this card are printed on-line.

The format of the Identification Card is as follows:

COLUMN	PUNCH
1	* (Produces BCD print ID)
2	* or blank (* if BCD off-line punching is done)
3	* or blank (* if binary off-line punching is done)
4	* or blank (* if 4020 tape is to be prepared)
5-7	maximum time in minutes
8-26	programmer's name and phone number
27-30	coder's number

COLUMN (continued)

31-33	name of code
34-36	group for which problem is done
37-38	category number
39	2 (for IVY codes)
40	G if debugging, H if production
42	machine used (Local conventions are used)
44	0
45	0
46	0
47-48	number of tapes used exclusively by and for this code*
73-80	programmer's name

MCP control cards. These cards may be required only in decks run on the 7030 if IVY is run under MCP. In this case these cards must precede any deck run on the 7030. These cards may be included in any IVY deck on any machine, however, and if not needed, will be ignored.

The purpose of these cards, all of which have a "B" in column 1, is to define input-output units in a symbolic manner; MCP then assigns absolute units to these symbolic numbers well in advance of the time the program is run, so that tapes can be mounted properly, etc. These cards

*The systems tape, standard print output tape, etc., are not included in this count.

must be the first ones present in any IVY deck which is run on the 7030; and, as mentioned above, can be removed for 7090 runs if desired.

The various "B" cards required are as follows:

A. Job Card.

```
1|2     9|10
B|       | JØB, IDENTIFICATION
```

Any identification desired, e.g., name and phone number, can be placed after the operation "JØB." Local conventions must be observed.

B. Type-of-Problem Card.

```
1|2     9|10
B|       | IVYGØ,
```

This card merely specifies that the problem coming up is in IVY language and will assemble and go.

C. Input-Output Definition Cards. One of these cards must be entered for each tape unit the programmer uses outside the system, i.e., for tape units other than the standard input-output tapes used by IVY. The format for these cards is as follows:

```
1|2      9|10                                                      68|69
B|IØDNAME | IØD,TAPE,EXIT,CHANNEL,NUMBER,MØDE,DENSITY,DISPØSITIØN|REF
```

where:

1. "IØDNAME" represents any symbol of from one to six alphabetic characters, used by MCP to be synonymous with the input-output requirement stated on the card.

2. "IØD," "TAPE," and "EXIT" occur as illustrated.

3. "CHANNEL" is any symbol from one to six alphabetic characters in length specifying some channel, the absolute address of which is assigned by MCP. Different symbols will be assigned different channel addresses. If the "CHANNEL" field is null, it is assumed that the channel

assignments of tape units are irrelevant, and MCP assigns any free tape unit regardless of channel.

4. "NUMBER" is the IVY tape number in hexadecimal (1,2,3,...,9,A, B,C,D,E,F). See page 120.

5. "MØDE" may specify either "ØDD," for odd parity, or "ECC," for odd parity plus ECC checking.

6. "DENSITY" is either "HD," for high density, or "LD," for low density. This must agree with the density, if any, requested for the tape in the calling sequence to "$TP," the tape program. See page 122.

7. "DISPØSITIØN" may be "NSAVE," for "do not save tape reels in any case;" "CSAVE," for "save tape reels only if job is complete;" "ISAVE," for "save tape reels only if job is incomplete; or "SAVE," for "save tape reels in any case."

8. "REF" is an octal number corresponding to the hexadecimal tape number in 4 above.

D. Reel Cards. A reel card must immediately follow the "IØD" card to which it refers, or another reel card referring to the same unit and channel. The format is:

```
1|2    9|10
B|      |REEL, R1,R2,...,etc.
```

where

1. "REEL" is the pseudo-operation defining this type of card.

2. "R_i" represents a symbol up to eight characters in length; the first three are not part of the reel identification, but specify whether the tape is labeled or not and whether the tape is protected (ring out) or unprotected. The remaining 5 characters agree with the identification shown on the physical reel. Thus R_i may be:

PLB xxxxx	protected, labelled
PUL xxxxx	protected, unlabelled
NLB xxxxx	unprotected, labelled

If an "R_i" is null, a labelled, unprotected tape is assumed. An "R_i" must be entered for each reel of the tape desired, even if only one reel is used. All reels are labelled automatically by MCP. For further details, the reader is referred to the MCP manual.

In general the programmer need not worry about punching the MCP control cards, since the 7030 run request sheet is used by the operators to punch the necessary "B" cards. These cards are placed in front of the deck, which is then run. Local conventions are important in the use of these cards and should be studied by the programmer interested in running on the 7030.

Start Card. A "start card" must precede every IVY code, behind any "*" or "B" cards. This card performs the following functions:

1. Erases the symbol table of the previous code, if any, and initializes IVY for a new code in such ways as setting base addresses for loading to initial values, etc.

2. Sets the print trigger on, which causes the contents of the start card and all cards following it to be printed on-line, until a "print suppress" card is encountered (see page 30).

3. Defines the maximum number of formulas in one formula set, numbers of independent store-address expressions, references within formulas, index registers, and numbered remark symbols used by the code.

The format of the Start Card is as follows:

Col. 1	Col. 2-72
S	(N_1), $A(N_2)$, $L(N_3)$, $X(N_4)$, SYMBØL$_1(N_5)$, SYMBØL$_2(N_6)$,...

Here "N_i" represents a decimal number which cannot be symbolized, and "SYMBØL$_i$" represents any legal symbol (from one to six alphabetic characters except the symbols A, L, or X). The "S" card cannot be followed by a continuation card.

The fields on this card will now be explained, with page references, where necessary, to indicate where further discussion of the concepts introduced by consideration of this card may be found:

1. (N_1): N_1 is the maximum number of formulas which will appear in any formula set of the IVY code introduced by the "S" card. Briefly, IVY codes are always divided into one or more subsets called formula sets, and each formula set contains one or more subdivisions called formulas. Within a formula set, the code can flow at will among the formulas, but direct branching between a formula in one formula set and a formula in another set is not allowed. Formula sets are to be thought of as almost independent packages of a code, to be entered from another formula set only by branching to the start of the set, and not to one of its formulas. (For further discussion, see page 114). Thus, if M_1 is the number of formulas in the first formula set, M_2 in the second set, ..., M_N in the nth set, then $N_1 = \max(M_1, M_2, \cdots, M_N)$. This entry causes a table to be constructed, $(N_1 + 25)$ words in length, to aid the compiler in assigning addresses to branches between formulas. The minimum value N_1 can have is 0.

2. A(N_2): The symbol "A" is <u>always</u> reserved for the "store address" symbol, even if no "store address" expressions are used in a code. If no "store address" expressions are used, the entry A(0) must still be present on the "S" card. If, however, the coder wishes to use "store address" expressions (which are usually helpful when working with multi-dimensional arrays), "N_2" specifies the maximum number of independent "store address" expressions in any formula of the program. (For further enlightenment see pages 75-78.) This entry causes a table, N_2 words in length, to be constructed for the use of the compiler in setting up "store address" instructions in machine language.

3. L(N_3): The symbol "L" is <u>always</u> reserved for internal branch references (L1, L2,...) within <u>formulas</u>. If no internal branch references are used, N_3 = 0, and L(0) must occur on the "S" card. N_3 is the maximum number of internal references within formulas, i.e., if J_1 is the number of references in the first formula, J_2 in the second,..., J_M in the Mth, then $N_3 = \max (J_1, J_2,..., J_M)$. (For discussion of "L" entries see pages 97-102.) This entry causes a table(N_3 + 25)in length to be constructed, to aid the compiler in assigning addresses to branches referring to "L" entries. N_3 can be at most 512.

4. X(N_4): The symbol "X" is <u>always</u> reserved for index registers. "N_4" specifies the number of index registers used in the program. The first N_4 consecutive index registers, X1, X2,..., XN_4 must be used, <u>not</u> any combination of N_4 different registers. Regardless of the machine used, IVY index registers always modify by addition, or <u>appear</u> to do so;

furthermore, IVY index registers are always positive, and even on Stretch (except in longhand coding) must not take on negative values. This entry causes a table N_4 in length to be constructed to aid the compiler in simulating extra index registers, if N_4 happens to be larger than the number of index registers available on the particular computer, and a second table, also N_4 in length, for aid in the computation of index branches. N_4 must be at least 1 but may be no larger than 256. For more discussion on index registers see pages 89-93.

5. SYMBØL_1(N_5): The remaining entries on the "S" card are optional (the first four listed above are mandatory) and specify numbered symbols which may be assigned only to remark blocks or calling sequence blocks. The symbol specified may be any symbol consisting of from one to six alphabetic characters except the symbols A, X, and L, which, as noted above, are always reserved for special purposes. "N_5" is the number of symbols which will begin with the alphabetic characters and end with one of the numbers 1, 2,..., N_5: SYMBØL1, SYMBØL2,..., SYMBØLN_5. Each of the blocks corresponding to these <u>numbered</u> symbols must be <u>loaded</u> separately on "R" or "K" cards (see below). A group of numbered symbols beginning with the same alphabetic symbol must all address the same type of information; that is, the symbols of a numbered block R, namely R1, R2,..., RN, must all address <u>either</u> remarks <u>or</u> calling sequence information, but <u>not</u> both. The number of numbered symbols allowed is obviously restricted to the remaining columns of the "S" card, since no continuation is allowed.

The discussion of the "S" card is now complete. Again it must be emphasized that the "S" card must not be followed by a continuation card, i.e., another "S" card or a card with the first column blank. All necessary information must be included on the one "S" card. The reader should also note that much of the information discussed above will be discussed in detail later. As a man once remarked when presented with the IVY system, "The 'S' card is supposed to be the first card in your code, but it will probably be the last one you write down on the coding sheet." The lesson here is clear: although an IVY deck must be ordered in a specific manner, quite often the order of coding will not correspond to the order of the deck, _or_ to the order of treatment of topics in this manual.

Print cards. These cards, the purpose of which is to turn the print trigger on or off, may occur anywhere in an IVY code. If the print trigger is on, all cards will be printed until it is turned off, thus allowing the coder to obtain a listing of all or part of this program. A card with "P" in column 1, and column 2 blank, turns the print trigger on; "S" in column 2 (for "suppress") turns the trigger off. Recall, as remarked on page 26, that an "S" card also turns the print trigger on. Thus, once a listing is obtained, on subsequent runs a "PS" card should follow the "S" card to suppress any unnecessary listing. The listing will appear off-line unless key 35 is down (7090) or binary key 63 is down (7030).

Comment cards. The "comment cards," which may occur anywhere whatsoever in an IVY deck, are announced by a "C" punch in column 1.

These cards are ignored by IVY for assembly purposes, except that their contents will be printed if the print trigger is on. Any printable comment may be punched on a "C" card; generally, of course, these comments are of an informational nature, describing the subsequent code for the benefit of anyone (including the coder) who might want to read it. "C" cards may be followed by any number of continuation cards with a "C" or "blank" in column 1.

Definition cards. After the "B," "*," and "S" cards the "definition cards" must occur. These cards, which have a "D" punch in column 1, are used to define symbols for data blocks, parameters, and formula sets. Formula names, however, should not be defined on "D" cards; these symbols are defined by their occurrence on "I" or "L" cards, described in Chapters 4 and 5 and in Appendices 2 and 3. Recall the distinction between formulas and formula sets, discussed previously on page 28. A detailed description of the allowed formats on "D" cards is given in Chapter 3, pages 43-57 . "D" cards may be followed by any number of continuation cards with a "D" or "blank" in column 1.

Remark cards. "Remark cards" provide a means for entering BCD information into core for printing comments on a listing, punching comments on cards, or for use as format statements for printing. Ordinarily remark cards should occur next after "D" cards in an IVY deck. Symbols may be defined on remark cards, remark blocks may be loaded, or a block of fixed length may be set up so that a remark may be constructed in it later using the character manipulation program described in Chapter 6, pages 154-156. Remark blocks may be named with numbered symbols entered on the "S" card (page 29) or with ordinary alphabetic symbols which have not been previously defined otherwise. A description of the format of remark cards occurs in Chapter 3, pages 61-64 . Remarks for use as format statements are described in Chapter 6, pages 132-145. The first card of a remark must have an "R" punch in column 1, because it is on this card that the symbol is defined; continuation cards, if any, are allowed, and must have a "blank" in column 1.

Calling sequence cards. "Calling sequence cards" are used for entering calling sequence information into core; calling sequence information may also be entered directly on instruction cards. However, the option of using calling sequence cards is allowed because of the flexibility of such a system: like remarks,calling sequence blocks can be defined without being loaded, so that values for them can be computed later in the code (see pages 182-184) for examples. Variable calling sequences, or calling sequences whose length depends on a parameter, may be defined; and a previously defined and loaded calling sequence can easily be altered. None of these operations is possible with calling sequences which occur on instruction cards. Discussions of the usage of calling sequences occur throughout this manual, e.g., Chapter 5, pages 105-110, and Chapter 8, pages 176-182. Calling sequences for particular IVY subroutines are discussed in Chapter 6. The actual format of calling sequence cards is described in Chapter 3, pages 64-67. As with remark cards, continuations of calling sequence cards must contain "blank" in column 1; the first card, on which the symbol of the block appears, must have a "K" punch in column 1. Unlike remark cards, however, the symbol on a "K" card must have been previously defined, i.e., entered into the symbol table; that is, it must be either a numbered symbol defined by an entry on the "S" card, <u>or</u> an alphabetic symbol defined by its appearance on a "D" card. (See Chapter 3, pages 47-48.)

<u>Instructions to operator card</u>. Cards of this type contain an "Ø" in column 1 and may not be followed by continuation cards. The "Ø" card, in columns 1 to 72, may contain any comment, interpreted as an instruction to the operator. When an "Ø" card is encountered by the loading program, it causes the following to take place:

1. The contents of the "Ø" card are printed on-line (using the printer on the IBM 7090 and machines without a typewriter, using the typewriter on machines which have one attached, such as the IBM 7030).

2. The machine then stops or waits, and a gong is sounded on machines which have one attached.

3. The operator presumably reads the instructions, carries them out, and presses an appropriate button ("start" on the 7090-type machines, "console signal" on the 7030), and IVY regains control and proceeds.

If the coder's program currently has control, the same functions may be performed by using the IVY subroutine "$ØP", described in Chapter 6, pages 151-152.

<u>Tape control card</u>. The purpose of the "tape control card" is to allow the programmer to read or write information on a binary, high- or low-density tape under control of the loading program. The same thing may also be done internally by using the IVY subroutine "$TP," described in Chapter 6, pages 121-126.

A tape control card has a "T" punch in column 1, and continuation cards, if any, must have a "blank" in column 1. The "T" card contains a calling sequence to ∅TP, consisting of various items of information separated by colons. These items are as follows ("H" is a hexadecimal digit, $1 \leq H \leq C$ on the 7090, $1 \leq H \leq F$ on the 7030):

ITEM	MEANING
∅HDH	set tape "H" to high density
∅LDH	set tape "H" to low density
∅RWH	rewind tape "H"
∅EFH	write end-of-file on tape "H"
∅ULH	rewind and unload tape "H"
∅ETH	write end-of-tape record, tape "H"
∅BBH,P	backspace tape "H" through "P" records
∅BFH,P	backspace tape "H" through "P" files
∅FBH,P	forward space tape "H" through "P" records
∅FFH,P	forward space tape "H" through "P" files
∅RDH,AD(∅WA) +P:AE(∅W)	read from tape "H" the record with ID = C(AD(∅WA)+P) into block AE
∅WRH,AD(∅WA) +P:AE(∅W)	write a record on tape "H" with ID=C(AD(∅WA)+P from block AE
∅RDH,AD(∅WA)+P	last entry only: compare ID of current record on tape "H" with contents of AD(∅WA)+P. "∅CS1" is set to 0 if not equal, 1 if equal.

In all of the above, "P" stands for "parameter algebra," which is explained at the beginning of Chapter 3. Other notation is explained in Chapter 5, pages 105-110, and the calling sequence for "$TP" is fully discussed in Chapter 6, pages 122-125. Page 120 contains a table showing correspondence between the tape number "H" as used above and tape and channel numbers on the 7090 and 7030. Below is an example of a "T" card and its continuation, which writes two blocks on tape, and reads in a third from another tape:

```
T|$RW3:$WR3,GE($WA)+1:SN($W):$WR3,AX($WA)+3:ST($W):$RW3:
 |$RW2:$FB2,4:$RD2,FNP($WA)+GE:FRNB($W):
```

Assembly card. The "assembly card," which has an "A" punch in column 1, and for which no continuations are permitted, is required to be present in an IVY program. Once this card is encountered it is assumed that all symbols (except those for formulas) have been defined, on "D," "S," and "R" cards. The purpose of the "A" card is to cause the instruction and/or longhand cards which follow it to be compressed and written in a specified file of a specified tape, or to read in and assemble instruction and/or longhand cards which have been previously written by an "A" card. Note that the "A" card differs quite markedly from the "T" card: The "T" card is used for writing or reading data; the "A" card is used to control assembly, and writes only unconverted instructions, and when reading, converts simultaneously into machine language. The use of "T" cards is optional, whereas "A" cards are required in order for the assembly to proceed properly. The two formats for "A" cards are as follows:

1. preceding code: The card

A|$WRN,F

causes the instructions on cards following to be written on tape "N," where "N" is a hexadecimal digit ($1 \leq N \leq C$ or F; see "T" cards), in the file number specified by "F," a decimal number. If N = 0, a special systems tape is used, equivalent to N = A.

2. following code: The card

A|$RDN,F

causes the unconverted code in file "F" of tape "N" to be read into core and converted to machine language.

An "A" card of type "1" will write instructions on tape until another "A" card or an "X" card (see below, pages 39-40) is encountered. The smallest unit of code which may be written using "A" cards is a <u>formula set</u>. In general it is best to write a long code in as many files as possible, one formula set per file, since, if several files contain part of the code which have been debugged, it is not necessary to read in the cards again for these particular files. One need only re-write and re-load the undebugged portions of the code; the rest may be read from tape using A|$RD cards. A completely debugged code may be read and assembled in its entirety from tape. Note that in re-writing a portion of the code which occurs in a certain file, it cannot be re-written in the same file (unless it occurs in the last file on tape) without destroying some subsequent information; it must be rewritten in a file beyond the last previously written file. Tapes written under the control of the "A" card may not be used interchangeably among machines using the same types of tape units (e.g., the 7090 and 7030) since they contain partially assembled code. Files of a tape may also be read and assembled under program control, using the IVY subroutine "$\not{S}$AP," described in Chapter 6, pages 119-120.

Instruction cards. "Instruction cards," which have an "I" in column 1 and may be followed by any number of continuation cards with "I" or "blank" in column 1, are used to load IVY algebraic code. The format of these cards, and the IVY algebraic language itself, are discussed in Chapters 4 and 5. "I" cards must be preceded by "A" cards, writing units of the code containing one or more formula sets on tape, and may be followed by other "A" cards or "X" cards, as described below, pages 119-120. All symbols for data, remarks, etc., must have been defined by the time the first instruction block is assembled, regardless of whether or not the block in question refers to these symbols, since instructions are loaded into core immediately above the symbol table; if the table has not been completed, subsequently defined symbols will destroy the first instructions of the code. Blocks of "I" cards must not contain "D," "R," "K," "T," "Ø," or "E" cards; in other words, all cards containing information not pertinent to instructions and their assembly must have been loaded before any instruction blocks, or must be loaded after the first "X" card.

Longhand cards. These cards, which have an "L" punch in column 1 and may be followed by any number of continuation cards with "L" or "blank" in column 1, are used for the entry of longhand instructions for a <u>particular</u> machine, as opposed to the "I" cards which enter the algebraic instructions valid on <u>all</u> machines. The formats of "L" cards are described in the appendices appropriate to the machines under consideration.

<u>Binary deck cards</u>. These cards, identified by an "F" in column 1, are used to load a relocatable column binary deck. The relocatable cards must, of course, contain instructions in the set of the particular machine being used, and must be in the proper relocatable format for that machine. The chief purpose of the "F" card is to allow a programmer to load a previously coded subroutine, <u>not</u> a complete code. The format of the "F: card is as follows:

```
F|:AD:$B,M:$A,L
```

> "AD" is a symbol for the <u>formula</u> <u>set</u> represented by the binary deck; "M" is the number of words (if any) reserved for data before the subroutine, in decimal; and "L" is the number of words (if any) reserved for data <u>after</u> the subroutine, in decimal. The purpose of the latter two entries is to take care that space is allowed for data blocks used by the subroutine for which no cards are loaded, such as, for instance, blocks defined by use of "BSS" or "BES" in the SAP and FAP systems. This is not necessary ordinarily on the 7030, since space for blocks defined by "DR" or "DRZ" is reserved by the use of special conventions on the binary cards.

Continuation cards are obviously not appropriate for "F" cards: IVY assumes that the cards following the "F" card are relocatable binary cards with 7 and 9 punches in column 1, and that the 1st non-relocatable card following is an IVY card with a non-blank punch in column 1.

The "F" card has been included primarily as a feature intended to simplify the transition from other coding systems to IVY; thus, subroutines available in relocatable form can be loaded in this manner until such time as they become available in IVY language. In no sense is IVY to be considered merely a relocatable loader: IVY recognizes only relocatable cards, and none of the other types of the large class of cards handled by the FORTRAN BSS loader.

It is the programmer's responsibility, then, to set up calling sequences to these relocatable routines correctly in the IVY language. Normally such subroutines should be self-contained, i.e., they should not refer to other subroutines, and should carry with them their own data and erasable blocks. If this is not done, then the programmer must exercise extreme care in the use of the subroutine. FØRTRAN, which can be used to produce relocatable routines which refer to outside data blocks and to other subroutines, stores data backwards in memory, at the time of this writing, while IVY stores data forwards. This difference should always be borne in mind when using a routine produced by FØRTRAN.

Execute card. The "execute card," with an "X" punch in column 1, is the IVY transition card; its detection causes IVY to transfer control to the programmer's code. Its format is as follows:

X|AD

where "AD" is the symbol for a formula set which must have been converted by means of an "A" card (or the routine "$AP") before the "X" card is encountered. If columns 2-72 of the "X" card are blank, it is assumed that the programmer has entered the loading program "$LD" from his code, and control is returned to the first instruction following the program's "$ LD" calling sequence. An "X" card with columns 2-72 blank is illegal if the programmer has not entered "$ LD" from his code. Normally, "$ LD" is entered to read data from "E" cards, described below.

Enter data cards. These cards, with an "E" punch in column 1, may be followed by any number of continuation cards marked by "E" or "blank" in column 1, and are used to enter data in blocks which have been previously defined on "D" cards. Normally "E" cards occur after the program's first "X" card, which transfers control to a specified formula set; "$ LD" is then entered to read the "E" cards, which must be followed by an "X" card with blanks in columns 2-72 to return control to the "$ LD" calling sequence. The format for "E" cards is described in Chapter 3, pages 57-59.

*Summary. It has been the intention of this chapter to describe the various types of cards used in an IVY deck, and as much as possible the order of discussion of these cards has been the order of an IVY deck at loading time. When possible, the card format has been described; in many cases, however, the reader has been referred ahead to those portions of the manual which describe the format of the card in question in more detail than can be attempted this early. In setting up an IVY deck for assembly, the programmer should keep one idea paramount: that IVY is a load-and-go, one pass system, meaning that every card is examined once and only once. Therefore, the order of loading is somewhat restricted in that symbols must be defined prior to their occurrence in code and calling sequences, making it necessary to place the "S," "D," and "R" cards in that order at the beginning of the deck. All symbols must be defined, i.e., entered in the symbol table, before any "K," "I," "L," or "F" cards occur, since the information loaded from these cards occupies space immediately above the symbol table, and any subsequent attempts to define symbols (treated and detected as errors) would destroy part of the information loaded by these cards.

*Because of the distinction between formula sets and formulas, as sets and subsets of a program, formula set names must be defined on "D" cards, whereas formula names are defined by their appearance on "I" or "L" cards and should not be defined on "D" cards. Thus, for instance, subroutines referred to by a number of formula sets should be defined as formula sets, since formulas can refer only to formula sets or to other formulas within the <u>same</u> set. All "I" or "L" cards must be written on tape and assembled using "A" cards, the usual procedure being to write each formula set in a separate file. This makes it possible to avoid reloading the entire deck for a second assembly, when none, or only a few, of the formula sets contain errors.

*Finally, after assembly of one or more formula sets, the "X" card transfers control to one of these sets and execution of the coder's program begins. At any time the program can re-enter "ȾLD" to load new data from "E" cards and regain control from an "X" card with columns 2-72 blank. The program also may use other IVY subroutines, such as "ȾAP" to assemble a new formula set, and various input-output routines for printing, punching, and the manipulation of tapes.

*Table I gives a summary of card types for quick reference, giving page numbers of descriptions and other useful information.

TABLE I

Table of Card Types

COL. 1	FORMAT ON PAGES	CONTINUATION ALLOWED?	PURPOSE
*	22-23	NO	Identification of off-line output
B	23-26	NO	Assignment of I/O on 7030
S	26-30	NO	Start, define essential quantities
P	30	NO	Set print trigger on or off
D	43-57	Yes "D" or "blank"	Define and/or load symbols
R	61-64	Yes "blank" only	Define and/or load remarks
K	64-67	Yes "blank" only	Define and/or load calling sequence blocks
Ø	33	NO	Instructions to operator
T	33-35	Yes "blank" only	Tape manipulation under loader control
A	35-37	NO	Write or read and assemble instructions
I	68-117	Yes "I" or "blank"	Load algebraic instructions
L	Appendices 2,3	Yes "L" or "blank"	Load longhand instructions
F	38-39	NO	Load relocatable binary deck
X	39-40	NO	Transfer control to program
E	57-59	Yes "E" or "blank"	Load data
C	30-31	Yes "C" or "blank"	Comment

CHAPTER 3

DEFINITION AND LOADING OF DATA, REMARKS, AND CALLING SEQUENCE BLOCKS

Definition and loading of parameters. A parameter, as referred to throughout this manual, is defined as a fixed point integer, the value of which remains constant throughout an assembly, and which is used to define such things as the dimensions of a block, conditions on whether assembly or loading of a block is to take place, and so on. The value of a parameter may, of course, vary from one assembly to another, but once defined for a given assembly, it must remain constant throughout the assembly. Since the notion of a parameter is the foundation of the whole IVY system, and the algebra of parameters is a cornerstone, the definition of parameters, followed by a discussion of parameter algebra, shall occupy us first in this chapter.

Since, as a rule, the entire assembly depends on the values of parameters, these quantities should be defined on the first "D" card or cards after the "S" card. In different assemblies these "parameter cards" can be changed for another set in order to change the dimensions of various arrays, change some of the conditional assembly statements, etc. Some simple parameter definitions are illustrated below:

D|GE = 2, AX = 15, BS(2) = 1,3,TH = 6, FINT(BS2) = 5, 6, 12

The first two symbols are defined as single parameters, the numbers 2 and 15. BS is defined as two parameters 1 and 3. When any block, parameter or not, is defined as being a vector or array N in length, N numbers must follow to load the block completely. More about this point later. TH is then singly defined, and finally FINT is defined as having length BS2, which is 3, and three numbers are loaded. Note that in the case of the parameters and data, the nth element of an array AD is addressed by writing ADn, where n must be a number and cannot be symbolized. However, the first element of a block may be addressed by using the symbol with no number, so that, using the above example, one may call on the number 2 by writing GE instead of GE1, though the latter is also allowed. Similarly the symbol BS alone would address the number 1, the symbol FINT alone would address the number 5.

Dimensions of multi-dimensional blocks can be symbolized by parameters defined in the above manner, or may be defined by fixed point numbers when dimensions never vary, or by parameter algebra, discussed below. New parameters may also be defined in terms of previous parameters, numbers, or parameter algebra involving previously defined parameters. Examples of this appear in the next section.

Parameter algebra. Parameter algebra is defined as fixed point integer algebra free of parentheses. The operations in this algebra, as in IVY "machine algebra" discussed in Chapter 4, take place in sequence from left to right, unmodified by parentheses. Examples of this algebra occur below, after a discussion of allowed operands and operations for this algebra.

The allowed <u>operands</u> in parameter algebra are:

1. Symbols which have been previously defined as fixed point integer parameters, e.g., GE, AX, and TH in the above example.

2. Symbols with a number, meaning the <u>nth</u> element of a previously defined fixed point integer parameter, e.g., BS2, FINT3 in the above example.

3. Literals, i.e., fixed point integers not symbolized, e.g., 2, 251, 3, 17, 23.

The allowed operations in parameter algebra are:

+	add
-	subtract
*	multiply
/	divide and truncate result to integer
+$	take absolute value of the preceding
-$	take negative absolute value of the preceding
*$	change sign of preceding
.$U	if result of preceding calculation is non-zero, set to 1
.$V	if result of preceding calculation is zero, set to 1; otherwise, set to zero.

Some examples of parameter algebra, involving the parameters defined in the example in the previous section, are as follows:

EXAMPLE	RESULT
TH + 3	9
AX + GE*BS2	51 (multiplication by BS2 times AC+GE: operations from left to right)
AX/TH+2	4 (result of division is 2)
AX+BS2/TH	3
-2*TH+AX.$\not{S}$U	1
-2*TH+AX.$\not{S}$V	0
FINT3*FINT-AX*$\not{S}$	-45
TH-AX+$\not{S}$	9

Examples of definition of new parameters using parameter algebra involving previously defined parameters:

```
D|AMP=GE-AX*BS2,NTT=AMP+3+$, PRT(GE+1) = AX*TH, 0, FINT2-BS2/GE,
```

Thus we note that the value of a parameter, as well as the dimensions of a block containing more than one parameter, can be defined by using parameter algebra involving previously defined parameters. Other examples of parameter algebra will occur in examples following treatment of the definition and loading of data and remark blocks.

The definition of symbols and loading of data. As remarked in Chapter 2, the definition of symbols (simply by their occurrence) and the loading of data may both be accomplished on "D" cards. One example of both symbol definition and loading is the case of parameters discussed in the previous section. We now come to the section covering the definition of other symbols without any loading being associated, as well as the definition of data blocks whose length may depend on previously defined parameters, and finally, the loading of these data blocks, which may occur from "D" or "E" cards. Data blocks may, of course, be left empty, to be filled by results calculated in the programmer's code.

Entries on "D" and "E" cards are separated by commas. Since continuation cards are allowed for both "D" and "E" cards, an entry may be continued from one card to the next; however, certain rules must be observed in this continuation:

1. Symbols and literals (i.e., numbers) cannot be continued from one card to the next, but must be complete on one card.
2. Entries within parentheses may not be continued from one card to the next, but must be complete on one card, including the right parenthesis.

For the moment, these two simple rules will suffice. Note that parameter algebra may be continued from one card to the next, providing that symbols and literals are not split, and that the algebra is not within parentheses.

Symbol definition. A symbol which occurs by itself between commas on a "D" card is placed in the symbol table, and thus defined. No address or other information is attached to the symbol table entry. It is in this manner that the names of formula sets and non-numbered symbols for calling sequence blocks must be defined. Example:

```
D|AGM, TDMT, LØGIC, FSA,FSB, FSC,
```

As was remarked on page 19, symbols consisting of a single alphabetic character need not be defined in this manner, since IVY always contains a table of the single character symbols.

Array definition. Arrays are defined on a "D" card by the appearance between commas of the symbol for the array followed by one or more (up to fifteen) parameter algebra expressions for the dimensions, enclosed in parentheses and separated by commas. No data are loaded for a block defined in this manner; however, an address is assigned and space is set aside for the array, which is now tagged as "data" in the symbol table. Example (using parameters defined in earlier examples in this chapter):

```
D|AVECT(NTT), BMULT(3,GE+1, 2*TH), CVEC(5), DMUL(2,5,GE),
```

In this example we note that the dimensions of an array can be defined by symbols, literals, or parameter algebra. The advantage of being able to symbolize the dimensions of an array is that by defining parameters properly, an array can always be assembled with the exact dimensions needed in a particular run. FORTRAN and similar systems do not allow array dimensions to be symbolized, and hence the programmer must allow space for the maximum size of an array, sometimes leading to storage problems, since usually all arrays do not simultaneously assume maximum size: one array may be smaller when another is larger. In IVY no such problem exists. By symbolizing dimensions, array sizes can be tailored to fit the particular input being used. In examining the above example, and looking back in the chapter to the examples on parameters, we see that AVECT is a vector 36 numbers long, BMULT is a $3 \times 3 \times 12$ array, CVEC is a vector of length 5, DMUL is an array 2 × 5 × 2 long.

In the event that one or more of the expressions for the dimensions of an array is zero, the array has length zero. A block legally defined in this manner is called a suppressed block. A block may be suppressed, for instance, when it is not being used at all in a particular assembly. When this is done, no error indication is given, and the assembly proceeds, replacing references to the block with references to the location of zero, and suppressing any "store" references to the block. The assumption is that since the block is suppressed, the portion of the code containing references to it will not be executed anyhow, or that replacement of the symbol by the address of zero is acceptable. Of course, in subsequent runs the coder may re-define the parameters used in computing dimensions of the block so that it is no longer suppressed.

If one or more of the expressions for the dimensions of an array is negative, an error indication is given, since obviously an array cannot have negative length or a negative dimension. Any references to such a data block in the code will be replaced by transfers which return control to IVY.

Loading of data on "D" cards. In addition to defining blocks as described above, loading may also be specified on "D" cards, by following the symbol and its dimensions, if any, with an equal sign and a number of expressions which load the block completely. These expressions are separated by commas. In the section on parameters, we have seen a number of examples of this, for instance:

```
D|GE=2, AX=15, BS(2) = 1,3, TH=6, FINT(BS2)=5, 6, 12
```

Here the symbols are defined by their occurrence and then loaded with the number or numbers to the right of the equal sign, in this case fixed point integers. We have also noted that symbols for fixed point quantities can be loaded using parameter algebra.

Besides fixed point numbers and parameter algebra, an array can be defined using a variety of expressions. The general case can be symbolized as follows:

$$\text{SYMBØL}(P_1, P_2, \cdots, P_N) = Q_1, Q_2, \cdots, Q_M$$

where "P_i" represents parameter algebra for the <u>ith</u> dimension, and the "Q_i" are expressions which cause the block to be loaded completely. The "Q_i" may be any of the following expressions:

1. A fixed point integer, that is, a string of decimal digits, preceded, if desired, by a sign, and the value of which must be less than 2^{27} on the 7090, 2^{38} on the 7030. <u>E.G.</u>: 1265, 12, -15792132

2. Parameter algebra, that is, parentheses-free algebra involving fixed point literals and symbols for fixed point numbers.

3. Octal fixed point integers, defined by prefixing an octal integer with a "B" in parentheses. Once a symbol has been loaded by an expression of this type it can appear in a parameter algebra expression. Octal numbers as such cannot appear in parameter algebra because this algebra must be parentheses free. Octal numbers are restricted to the same magnitudes as fixed point numbers, given above. Once the "B" occurs, all numbers thereafter for the same array are considered octal until overruled by some other entry. Example:

```
D|XPL(2)=(B)77653,-62713, RST(3)=256,-7212,(B)1371,FNP=XPL2+769
```

In this case 77653, -62713, and 1371 are octal. 256,-7212, and 769 are decimal.

4. Boolean words, defined by prefixing an unsigned octal number with a "W" in parentheses. A Boolean word is used in logical or Boolean arithmetic and may fill the entire machine word; thus a Boolean word must be less than 2^{36} on the 7090 and 2^{64} on the 7030. Boolean words cannot be used in parameter algebra, but only in the machine algebra described in Chapter 4 (see pages 93-96). The prefix "W" operates in the same manner as "B", that is, all numbers entered thereafter for the same array are considered Boolean until overruled by some other entry. Example:

```
D|AXX(3)=(W)457620001713, 7625313, 963, AYX(2)=(B)76632, (W)75931,
```

In the above, 457620001713 and 762313 are Boolean, while 963 is fixed point, because it contains a digit greater than 7. In the loading of AYX we see the "(W)" overruling the "(B)" on the first entry. Note that Boolean numbers are always unsigned.

5. Fixed point decimal numbers may also be entered by prefixing them with "A" in parentheses, in the case where a "B" or "W" is operative and the fixed point number does not contain a digit greater than 7. Like the latter, "A" holds for the same array until overruled. Example:

```
D|AABC(3) = (B)70707, 17231, (A)26513
```

70707 and 17231 are octal numbers and 26513 is decimal.

6. Floating point numbers may be entered using the following sequence of characters: a sign (optional), a string of from 1 to 16 decimal digits containing a decimal point, followed by another sign and a fixed point number representing the exponent (optional). By "exponent" is meant the power of ten by which the expression is to be multiplied. For example:

```
D|AACCD(2,2) = 3.1415926535, -2.742653-7, 500.263+12, -21.732,
```

All the numbers above are legal floating point numbers. Floating point numbers N are restricted to approximately $10^{-38} < N < 10^{38}$ on the 7090, $10^{-307} < N < 10^{307}$ on the 7030.

7. Zeroes may be inserted by prefixing a parameter algebra expression with "Z" in parentheses. The number of zeros

specified by the algebraic expression is entered. If no parameter algebra is given, the remainder of the block is filled with zeros. For example:

```
D|ACDX(20,30)=2.7123, 5.7561, (Z), ARPX(NTT) = (Z)NTT-3, 5.23, 6.51, 7.32
```

Two numbers are entered in "ACDX" and the remainder of the block is set to zero. All but the last three locations of "ARPX" are set to zero, then the remaining three non-zero numbers are loaded. In both cases, loading is complete, as required.

8. A given number of locations may be skipped (without being set to zero) by the entry "S" in parentheses followed by a parameter algebra expression. The "skip" feature is written in the same manner as the "zero" feature. For example:

   ```
   D|ACDY(NTT) = 3, 6, 12, (S), ACDA(21) = 2.0, (S) 19, 3.561,
   ```

9. A number, once entered, may be repeated a specified number of times by following it with "R" in parentheses and parameter algebra telling the number of repetitions desired. As with "Z" and "S," if no parameter algebra is given, or ifthe result of the algebra is zero, the number is repeated until the end of the block. For example:

```
D|ACDB(5) = 2.7653+6, (R), ACDC(NTT)=2.5617, 9.986301-10, (R)NTT-3,8.653,
```

In "ACDB", the entire block is filled with one number; however, only a portion of "ACDC" is filled with the repeated number 9.986301-10. As always, loading is complete. The last N numbers loaded into a block may be repeated M times by the entry "N(R)M" between commas, as illustrated below:

```
D|BBCX(25) = 3,2,1,5,4 (R) 5,6.513, ...
```

The numbers 3, 2, 1, 5 are entered six times; the last number of the block is 6.513.

10. Any number of linear interpolants may be entered between two floating point numbers by placing between them an "I" in parentheses followed by parameter algebra specifying the number of interpolants desired. Note that this entry may be used only with floating point numbers. For example:

    ```
    D|ACDD(626) = 1.0, (I) 623,625.0, 7.363-11,
    ```

The 623 interpolants 2.0, 3.0, ..., 624.0 are entered in "ACDD" between the two numbers shown.

11. A block may be loaded with multiples of a fixed point number by the entry of "M" in parentheses followed by parameter algebra specifying the number for which multiples are desired. If the block has dimension "P," the multiples of a specified number "N" entered are: 0, N, 2N, 3N, ..., (P-1)N. Only the entry for multiples may occur if it occurs at all in the loading of an array. Examples:

```
D|ACDXM(30)=(M)20,BMULXA(5)=(M)2,DMULXB(GE)=(M)5,

D|BMULTA(GE+1)=(M)3,BMULTB(2*TH)=(M)GE+1,
```

The multiples defined by this example are the same as would be obtained by writing

```
D|ACDXM(30) = 0,20,40,60,...,580, DMULXA(5)=0,2,4,6,8,etc.
```

This example computes what we shall call the index multiples of the arrays ACDX, DMUL, and BMULT, which were defined in previous examples of this chapter. For further discussion see below, page 54, and examples in Chapter 8, pages 168-169. By use of the "(M)" entry one can also load the multiples of a number I plus a second number J. The entry"J(M)I" will enter the numbers J, J+I, J+2*I, J+3*I, etc., to the end of the block named. Thus, for instance, one might enter the 476 consecutive numbers 25, 26, 27, ..., 499,500 by the following entry:

```
D|CPDAL(426)=25(M)1,...
```

12. A group of floating point numbers all having the same exponent may be entered without writing the exponent more than once, by preceding them with an "E" in parentheses followed by the exponent, in fixed decimal representation (Parameter algebra is not allowed). For example, the following two entries are equivalent:

```
D|HFNT(6)=3.512+6,-2.713+6,9.916+6,20.251+6,-3.3216+6,2.515+6,

D|HFNT(6)=(E)+6,3.512,-2.713,9.916,20.251,-3.3216,2.515,
```

The exponent specified by the "E" entry is effective until it is overruled by a different "E" entry, a fixed point number, a floating point number with an explicitly stated exponent, or the definition of another symbol.

The usage of index multiples. Before we continue our jaunt through the jungle of "D" and "E" card notation, a brief aside on index multiples is appropriate at this point, to ease any curiosity on the subject that might have been aroused by paragraph 11 above.

Suppose we have an array "B," which has been defined with dimensions I, J, K. (The multidimensional case can be generalized from this treatment of the three-dimensional case.) In IVY the first element of this array will have indices (1,1,1). Most coding systems start indexing with (0,0,0) because of the way computers are built; but IVY, by an internal trick, causes all indexing to start with 1. Similarly the last element of this array has indices (I,J,K). Now, suppose we wish to compute the address of some random element (i,j,k) of the array. If "B" represents the base address minus 1, the address of the (i,j,k) element is:

$$B + i + (j-1)\ I + (k - 1)\ IJ$$

We see that to compute this address, three multiplications are necessary; in general, for an n-dimensional array, $\frac{n(n-1)}{2}$ multiplications are required to compute the address. However, multiplications can be avoided altogether if we happen to have access to a table of the multiples of I and of IJ. It is for this purpose that the "M" entry is used on "D" cards: to set up tables of index multiples for all arrays where random addressing is needed. The actual details of coding involving the use of index multiples are covered in Chapter 8, pages 168-169.

Other entries allowed on "D" cards. Besides defining symbols and blocks, with the options of loading mentioned above, two other types of entries are permitted on "D" cards: one to set up equivalent blocks, and the second to skip certain definitions under parameter control.

1. Equivalent blocks are two blocks of data which share the same memory locations and have the same dimensions, but which have different symbols. The use of equivalence is a means to conserve storage by using the same area for a second array when the need for the first has disappeared. A second block is defined as equivalent to a previously defined block by prefixing its symbol with an asterisk (*) and following it with an equal sign followed by the first symbol. The second symbol must not have been previously defined. A symbol which has once appeared on the left of an equivalence may not appear on the right of a subsequent equivalence statement; i.e., equivalence chains are not allowed. However, two or more symbols may be defined as equivalent to the same symbol. The example which follows is based on previous block definitions given as examples in this chapter:

```
D|*ACDE=ACDD, *ACDF=GE,*ACDH=GE, *ACDG=BMULT,
```

so that "ACDE" is a vector 626 in length, "ACDF" and "ACDH" both represent the same parameter "GE," and "ACDG" is a block with dimensions (3, GE+1, 2*TH) sharing storage with "BMULT."

2. The jump feature allows the programmer to skip certain definitions of symbols, or to define a symbol in one of several ways, under

parameter control. This is effected by placing "$\not{}$J" in parentheses, followed by a parameter algebra expression and an equal sign with one of eight conditions. If the condition is met, the definitions following the comma are skipped until another "ǿJ" is encountered between commas. The general format is as follows:

```
(ǿJ) P=C, ... (definitions) ..., ǿJ, ...
```

where "P" represents a parameter algebra expression and "C" represents one of the conditions:

C	CONDITION
Z or 0	jump if expression is zero
NZ	jump if expression is non-zero
LZ	jump if expression is less than zero
ZL	jump if expression is zero or less
GZ	jump if expression is greater than zero
ZG	jump if expression is zero or greater
P	jump if expression is plus
M	jump if expression is minus

If the condition is met, the definitions following the comma are skipped until the ",ǿJ," is encountered. If the condition is not met, the definitions are handled in a normal manner. For example:

```
D|(ǿJ) GE-2=0, ACDH(5,7,11),*ACDL=BMULT, ǿJ,
D|(ǿJ) 2*TH-7=ZG,ACDJ(2*NTT,GE+1),ǿJ,(ǿJ)2*TH-7=LZ,
D|ACDJ(2*TH-7, GE+1), ǿJ,
```

Here we see that the definitions of "ACDH" and "ACDI" are skipped if GE is equal to 2, and that the definition of "ACDJ" takes on one of two forms depending on the value of "2*TH-7."

The loading of data on "E" cards. Data may also be loaded from "E" cards, but one should bear in mind that all symbols appearing on "E" cards must have been previously defined on "D" cards, and the dimensions of any blocks loaded from "E" cards must also have been previously defined. Thus on "E" cards only the symbol for the block can appear to the left of an equal sign, since the dimensions are known. The expressions allowed on the right of the equal sign are the same as those allowed on "D" cards. For example, the following "E" card will load two blocks defined in examples of "D" cards earlier in this chapter:

```
E|BMULT = 2.56312-13, (I)25,9.6732153-2, (R)3, (S),CVEC=1.,2.,3.,4.,11.72,
```

Note that as on "D" cards, loading must be complete. The "(S)" in the expression for "BMULT" guarantees this.

The "jump" feature is also allowed on "E" cards in order to skip the loading of a certain block or to load it in one of two or more alternative ways. The "equivalence" feature is not allowed unless the symbol to the left of the equal sign has been defined by its appearance on a "D" card sans dimensions, since the "E" card is merely a loading card, and not one on which symbols can be defined. In other words, no symbol can make its first appearance on an "E" card. For example, the entry "*TIMT=BMULT" is permissible since "TIMT" has occurred on an earlier "D" card in this chapter, without dimensions. "*BMULT=AVECT" is not legal since "BMULT" has already been defined with dimensions and hence assigned a location in core. "*ACDK=BMULT" is not legal since "ACDK" has not occurred at all on a previous "D" card.

Double-stored data. An entry for loading double-stored data is permitted on "E" cards; this entry is not allowed on "D" cards. Double-stored data is data each word of which contains two numbers paired in the single memory location. The high-order portion, called the quantity or simply "Q," can be a signed, fixed or floating point number. The low-order portion, called the tag or "T," is an unsigned fixed-point integer which must be less than 2^{15}. The exact length of the tag in bits can be specified by parameter algebra. The IVY algebraic language has special formats for handling double-stored data, discussed in Chapter 4, pages 79-80 . Double-stored data has many uses, the chief being in mesh-type problems for solving differential equations. For instance, in boundary value problems, the boundary points can be labeled with tags having different values from the tags at interior points. In hydrodynamics codes, points corresponding to different substances can be identified by their tags, and so on. The format for loading double-stored data is as follows:

$$\text{SYMBØL(Q.P)} = E_1, E_2, \ldots, E_M, \text{SYMBØL(T.P)} = F_1, F_2, \ldots, F_N$$

where the "Q" signals that the "Q" portion of the block is to be loaded, and "P" represents parameter algebra whose value gives the tag length in bits. The "E_i" are expressions for fixed or floating point numbers which completely load the block. "T" now signals that the tag portion is to be loaded, "P" being the same expression for the length of the tag as appeared in the first parentheses. The "F_i" are expressions for unsigned fixed point integers which completely load the block. For example:

```
E|AVECT(Q.3)=5.32132-2,4.71531-1,(I)NTT-3,2.1532+2, AVECT(T.3)=
E|(B)1,2,3,7,6,5,4,2,1,7,(R),BMULT(T.GE+2)=TH,TH+1,TH+2,15,14,
E|12, 3, 5, (S),
```

We thus note that AVECT is loaded with a tag 3 bits in length, and that both the "Q" and "T" portions are loaded completely. "BMULT," a block which has been loaded previously, is now defined to have a tag GE+2, or 4, bits long, and the tag portion is then loaded. This is possible only when a block contains floating point numbers: the block can be loaded on a "D" or "E" card with floating point numbers _without_ specifying "Q" and the tag length, and the tag can then be loaded on "E" cards in the normal way.

Summary. Before proceeding to the study of remark and calling sequence cards, a summary of the treatment of "D" and "E" cards is needed. This summary, for review purposes as well as for quick reference, is given in Table II.

TABLE II

SUMMARY OF ENTRIES ALLOWED ON "D" AND "E" CARDS

COL. 1	FORMATS AND EXPLANATION:
D	SYMBØL$_1$,SYMBØL$_2$ $(I_1, I_2,\ldots,I_N)$, SYMBØL$_3$ = Q, SYMBØL$_4$ $(J_1, J_2, \ldots, J_M) = E_1, E_2,\ldots,E_L$ 1. "SYMBØL$_1$" is entered in symbol table. 2. "SYMBØL$_2$" is entered in symbol table, assigned an address, and allotted $I_1\ I_2 \ldots I_N$ words of core. 3. "SYMBØL$_3$" is entered in symbol table, assigned address, and loaded with Q. 4. "SYMBØL$_4$" is handled as case 2 and also loaded. Loading must be complete. "Q" and the "E_i" may be any of the following: a. Fixed point number (sign and decimal digits) or parameter algebra. b. Floating point number (sign, digits with decimal point, ± exp. if desired). c. (B)N_1, N_2,...N_1, N_2, etc., are octal fixed point integers until (B) overruled. d. (W)M_1, M_2,...M_1, M_2, etc., are Boolean octal words until (W) overruled. e. (A)L_1, L_2,...L_1, L_2, etc., are fixed point decimal integers until (A) overruled. f. (Z)P insert P zeroes. "P" represents parameter algebra. Proceed to end of block if P=0. g. (R)P repeat last entry P times. h. (S)P skip P words. i. (I)P insert P interpolants between 2 fl. pt. numbers j. (M)P insert multiples of P (0, P, 2P, 3P,...,) to end of block. k. (E)±N the following fl. pt. numbers all have exponent = ± N until overruled. All names <u>except</u> remark names, numbered symbols and formula names must be defined on "D" cards.

TABLE II (Continued)

COL. 1	FORMATS AND EXPLANATION
D	$*SYMBØL_i$ = $SYMBØL_j$,($J) P=C,...definitions...$J 1. $SYMBØL_i$ equivalent to $SYMBØL_j$ providing $SYMBØL_j$ is defined and was not defined by another "*" statement. 2. If parameter algebra "P" satisfies one of the conditions "C" (Z or O, NZ, LZ, GZ, ZL, ZG, P, M) definitions are skipped.
E	$SYMBØL_1$ = P_1, P_2,...,P_N, $SYMBØL_2$(Q.P) = Q_1,Q_2,...Q_M, $SYMBØL_3$ (T.P) = T_1, T_2,..., T_K Used for loading previously defined blocks. Loading must be complete. 1. "P_i"and"Q_i"are any of the expressions under "4" above. 2. "T_i"are unsigned fixed point numbers. 3. "Q.P" and "T.P" mean "quantity" and "tag" of DS number, "P" = tag length in bits. Jump and equivalence can also be used on "E" cards, equivalence with some restrictions.

The definition and loading of remarks. Remarks are usually for one of two purposes: first, to provide comments and headings for output listings and cards; and second, to provide format statements for printing, punching, and microfilm output. In this section we will consider only remarks for comment purposes. Remarks for use as format statements are described in Chapter 6, pages 132-145. By using "R" cards and their continuations, if necessary, remark blocks can be defined and/or loaded; the "R" card for remarks is thus analogous in function to the "D" card for data. However, no great parallel is found in the way these cards are punched. An "R" card must always begin with a symbol which is called the "name" of the remark. Only one name can appear on an "R" card. If a remark is too long to fit on a single card, it may be continued on the next card providing the continuation card has a blank in column 1. (This point was first made in Chapter 2.)

The general format of an "R" card is as follows:

1. "R" in column 1.

2. Symbol or name of remark. This may be any legal symbol or numbered symbol.

3. Optional: after the symbol, a parameter algebra expression in parentheses. The value of this expression is taken by IVY to be the number of characters in the remark, so that IVY will set aside this amount of space for the remark.

4. An equal sign, followed by any group of Hollerith characters, which may fill any number of cards. The number of characters in the remark must not exceed the number defined by the entry of "3" if this option is used.

5. In three consecutive columns, the characters $\$\$\$$ to signal the end of the remark. $\$\$\$$ need not appear if the remark is terminated by the end of a card.

The remark as stored in core consists of all characters, taken in order, from the first character to the right of the equal sign to the last character to the left of $\$\$\$$. These characters will be printed if the symbol of the remark is specified in a calling sequence to "$PR," the print routine (page 131). By using the characters $\$\$$ in two consecutive columns in a remark, the remark may be printed on two or more lines: the portion of the remark following $\$\$$ will be printed on the next line below the portion preceding $\$\$$. The characters $\$\$$ are not printed. Examples of remark entries:

```
    REMARK                                                  NOTES
R | RL =ABUNDANCES^AND^ACTIVITIES^ØF^ISØTØPES$$$            1, 2
R | REM(NTT*3+GE) = $$$                                     3
R | R2 = THIS^IS^A^$$^TWØ^LINE^REMARK$$$                    4
R | R3 (3627) = CØNSTRUCTED^GRAPH^FØLLØWS$$^$$$             5
R | REMB=BESSEL^FUNCTIONS^COMPUTED^BY^RECURSION^FORMULA.^ SEE^ANY^TABLE^TO^V
  | ERIFY^ACCURACY. $$$                                     6
```

Notes:

1. The following conventions should be observed to make the coding sheets appear unambiguous to the keypuncher: "blank" is represented by the carat "^"; n blanks are represented by the number "n" in a box: $\boxed{n}$ Alphabetic "I" must be written with bars or dotted ("i") to distinguish it from numeric "1"; alphabetic "Ø" is slashed to distinguish it from numeric zero; and alphabetic "Ƶ" is slashed to distinguish it from number two.

2. Our first remark is an illustration of a simple heading, the name of which is a numbered symbol presumably defined on the "S" card.

3. In this case, space is assigned for a remark having NTT*3+GE or 110 characters. No characters are loaded. It is presumed that a remark will later be constructed in the space, for instance, by "$CM," the character manipulation subroutine described in Chapter 6, pages 154-156.

4. In R2 we see the $$ convention for printing a remark on two lines. If printed, this remark will appear as follows:

```
THIS IS A
TWØ LINE REMARK
```

5. Space for 3627 characters is reserved. 27 of these are loaded, namely, the comment and its $$ spacing control. Note that to avoid confusion, the $$ and $$$ are separated by a blank. 3600 spaces remain in a block; a 60 × 60 character graph could, for instance, be constructed in the remaining portion of this remark block using "$CM."

6. An example of a remark which is continued onto a second card. Note that the continuation card has a blank in column 1, as required.

The loading of calling sequence blocks. Calling sequence blocks, for the use of subroutines, are loaded from "K" cards. As has been remarked previously, the symbol assigned to a calling sequence block must have been previously defined, either by its occurrence on a "D" card or by its occurrence in the naming of a numbered block on an "S" card. Continuations of a "K" card must have "blank" in column 1 since the symbol of the block is assigned on the "K" card. The general format of a "K" card is as follows:

K|SYMBØL(P) = (...calling sequence information...)

where, of course, the calling sequence information enclosed in parentheses may be continued onto subsequent cards if necessary. "SYMBØL" represents any legal, previously defined symbol or numbered symbol. The optional entry "(P)" is parameter algebra defining the length of the calling sequence in machine words. Inside the parentheses to the right of the equal sign may occur any number of calling sequence word entries, separated by colons. The information in each entry between colons is stored into one full word of the machine being used. The calling sequence word entry, to be described shortly, allows for great flexibility in constructing calling sequences. The IVY subroutines described in Chapter 6 use only a portion of the available calling sequence words; however, the full generality is available for those programmers who wish to construct their own subroutines and calling sequences.

<u>A digression on notation</u>. Certain notations are used in the IVY system for addressing various quantities connected with the control word, i.e., the word associated with a symbol in the symbol table which contains the base address minus one and the count of the block having this particular symbol. These notations are as follows, where "AD" represents any symbol, except IVY symbols starting with "$":

NOTATION	QUANTITY GIVEN IN CALLING SEQUENCE
AD($W)	control word of "AD"
AD($WP)	position of control word of "AD"
AD($WA)	control word address (base address -1 of "AD")
AD($WC)	control word count of "AD"

Calling sequence word entries on "K" cards set up full words in the calling sequence in the same format as control words. Thus, below we will speak of the "$WA" or "$WC" portion of a calling sequence word, and it is hoped that as the manual progresses, the reasons for this notation will become more clear.

<u>Calling sequence word entries</u>. Calling sequences may contain any of the following entries between colons:

1. $XXX, where "XXX" represents one, two, or three alphanumeric characters. The core BCD (octal) equivalent of these characters, exclusive of the "$," is placed in the "$WC" portion of the calling sequence word. A table of hollerith characters and their BCD octal equivalents follows:

TABLE III

HOLLERITH CHARACTERS AND OCTAL EQUIVALENTS

Character	BCD	Character	BCD	Character	BCD
0	60	H	30	X	67
1	01	I	31	Y	70
2	02	J	41	Z	71
3	03	K	42	+	20
4	04	L	43	-	40
5	05	M	44	*	54
6	06	N	45	/	61
7	07	Ø	46	=	13
8	10	P	47	'	14
9	11	Q	50	.	33
A	21	R	51	:	72
B	22	S	62	$	53
C	23	T	63	(	74
D	24	U	64	)	34
E	25	V	65	,	73
F	26	W	66	blank	00
G	27				

2. AD($W)+P, where "AD" represents any legal symbol, numbered or not, and "+P" represents parameter algebra. The control word associated with "AD," as modified by parameter algebra, is placed in the entire calling sequence word.

3a. AD($WP), where "AD" represents any legal symbol, numbered or not. The location of the control word of "AD" is placed in the "$WA" portion of the calling sequence word.

b. AD($WC)+P, where "AD" is the above and "+P" is any parameter algebra. The count of the control word of "AD," as modified by the parameter algebra, is placed in the "$WA" portion of the calling sequence word.

c. AD($WA)+P. Same as "3b" except the address of the control word of "AD" is used.

d. AD(P), where "AD" is any legal <u>non-numbered</u> symbol and "P" is parameter algebra. The <u>contents</u> of location AD($WA)+P are placed in the calling sequence word.

e. P, i.e., parameter algebra. The result of the algebra is placed in the calling sequence word.

4a. An entry of type 1, followed by a comma and an entry of type 3a, 3b, <u>or</u> 3c is allowed.

b. An entry of type 1 followed by a comma and by an entry of type 3d is allowed, providing the number addressed is fixed point and less than 2^{18}, or by an entry of type 3e providing the result of the parameter algebra is less than 2^{18}.

Chapter 6 is rife with examples of calling sequences to the various IVY subroutines. Only one example will be given here: suppose we wish to enter "$TP," the tape program, internally. If we use the same calling sequence covered in the example in Chapter 2 (page 35) on "T" cards, assigning it the name "TAPE" (which we presume has been defined by its previous occurrence on a "D" card), the "K" card calling sequence appears as follows:

```
K|TAPE=($RW3:$WR3,GE($WA)+1:SN($W):$WR3,AX($WA)+3:ST($W):$RW3:
 |$RW2:$FB2,4:$RD2,FNP($WA)+GE:FRNB($W))
```

Note that, as on "R" cards, the parameter algebra expressing the length of the calling sequence is optional. If this algebra is given, the calling sequence need not be completely filled, and entries can be computed by the programmer, if desired. Examples of this technique appear in Chapter 8, pages 182-184.

CHAPTER 4

THE IVY ALGEBRAIC LANGUAGE

The IVY algebraic language, or machine algebra, is capable of handling expressions in floating point, fixed point, Boolean, or index register algebra, as well as the simple parameter algebra already discussed in Chapter 3. We will consider these types of algebra in succession, with examples. Index branching will be covered along with index algebra and Chapter 5 will consider other types of branching. A summary of machine algebra appears in Chapter 9, pages 190-195.

The operation "=" is permitted in all classes of algebra except parameter algebra, and means the following: "evaluate the expression to the right of the '=' sign, and place the result in the location specified on the left of the '=' sign." In other words, expressions such as

$$B = B + 1$$

are allowed and make sense with this definition of "=". This statement means "increment the number in location 'B' by one." With this preliminary remark we shall launch ourselves into a discussion of the various types of algebra.

Floating point algebra. The following operations are permitted in floating point algebra:

OPERATION	NOTE	MEANING
+		add
-		subtract
*		multiply
**	1	raise to a power
/		divide
//	2	reciprocal divide
+$		take abs. value of preceding
-$		take neg. abs. value of preceding
*$		change sign of preceding
.$R	3	take square root of preceding
.$CX		convert exponent minus one of preceding result to fixed point integer. (Integer part of $\log_2$ [result]).
.$CA		convert preceding result to fixed point integer.
.$U		if result of preceding is $\neq 0$, set to 1.
.$V		If result of preceding is = 0, set to 1; otherwise set to 0.

NOTES:

1. The expression for the exponent, following "**," must be in parameter algebra. For example:

   B**2, B**(GE + 1), B**(2-TH)

2. Reciprocal divide differs from regular divide only in that the denominator appears first. Thus, "C//B" and "B/C" are equivalent. See pages 72-73.

3. This operation is valid only when the preceding result is positive. If an attempt is made to take the square root of a negative number, an indicator is set which may be tested by entering the IVY subroutine "$TT" (Chapter 6, pages 128-130 .)

Parenthesis conventions. Floating point algebra, unlike parameter algebra, may contain parentheses, for one of two purposes: first, to contain a modifier of a symbol, or second, to contain units of the algebra. The second use will be described here; the first will be encountered later in the section on modifiers (pages 74-84).

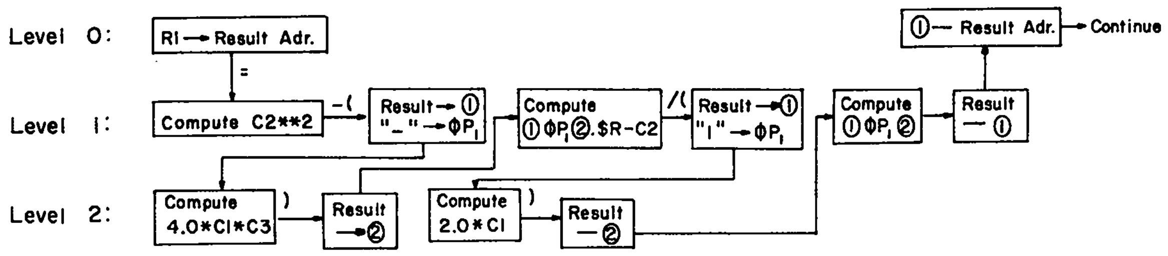
As in parameter algebra, if parentheses are lacking altogether, operations proceed in simple sequence from left to right. For example, observe the following equation in machine algebra without parentheses and its equivalent in display algebra:

MACHINE	DISPLAY
R1 = C2**2 - 4.0*C1*C3.$R - C2/2.0*C1,	$r_1 = \frac{\sqrt{(c_2^2 - 4)c_1c_3} - c_2}{2} \cdot c_1$

The equation shown is an attempt to use the quadratic formula to find one root of a quadratic equation with real roots. Correctly written, with parentheses, the equations appear as follows:

MACHINE	DISPLAY
R1 = C2**2-(4.0*C1*C3) .$R-C2/(2.0*C1),	$r_1 = \frac{\sqrt{c_2^2 - 4c_1c_3} - c_2}{2c_1}$

Note that the purpose of parentheses in machine algebra is to localize the operations so that they do not affect the result of the previous computation. When a left parenthesis is encountered, a new level of operation commences in which the algebra within parentheses is performed, with the convention that each operation within parentheses is performed on the preceding result only as far back as the left parenthesis. The above example might be diagrammed as follows:



Ten levels, i.e., ten sets of parentheses within parentheses, are allowed in IVY. Thus one can evaluate quite complicated expressions in the machine algebra, for example:

MACHINE	DISPLAY
2.0 + (C1*(C2**(3+TH))) +C3**2	$(2+c_1c_2^{3+TH} + c_3)^2$

Note: To avoid confusion, brackets "[ ]" and "curly" brackets "{ }" can be used on coding sheets if desired. They will be <u>punched</u> as parentheses, however.

However, it should be pointed out that this example can be written with only one set of parentheses, as follows:

C2**(3+TH)*C1 + 2.0 + C3**2

In general most equations can be optimized so that a minimum of parentheses occur, by moving multiplications and exponentiation to the beginning of the equation and by using the "reciprocal divide" (//) instruction. Below is the quadratic formula, optimized in this fashion, and evaluation of the polynomial $P_4 = d_1y^3 + d_2y^2 + d_3y + d_4$, in machine algebra:

```
R1 = 2.0*C1//C2**2-(4*C1*C3).$R-C2
P4 = D1*Y+D2*Y+D3*Y+D4
```

Thus, note that the quadratic formula cannot be optimized for better than one set of parentheses, while polynomial evaluation needs no parentheses whatsoever. A peculiarity of the "//" operation should be noted here: everything to the left of "//" is the denominator of the fraction being computed; everything to the right is the numerator. Thus parentheses are not needed to enclose either expression, and the field of the square root operation (in this case) need not be enclosed in parentheses since its field of operation is assumed to start to the right of the "//."

A further advantage of the "//" operation becomes apparent if one considers the evaluation of continued fractions; for instance, in display algebra, the expression

$$y = \cfrac{b}{h + \cfrac{c}{d + \cfrac{e}{f + g}}}$$

This quantity can be written in machine algebra in either of the following ways:

```
Y = ((F + G//E) + D//C)+H//B,       or Y = B/(H + (C/D + (E/(F+G))))
```

We note that the first expression, using the reciprocal divide instruction, has two less sets of parentheses than the second. Generally, algebraic expressions containing fractions with complicated denominators can be evaluated more efficiently using the "//" instruction. See Chapter 8, pages 163-164, for further discussion.

Operands which may appear in a floating point expression. We have seen above a number of allowed operands in the examples given. Below is a complete list of the operands which may appear:

1. Any symbol for a single word or array which has been defined and/or loaded as a floating point number, interpreted by IVY to mean the first element of the block.

2. Any symbol as in "1" followed by a number n, interpreted by IVY to mean the nth element of the block.

3. Any floating point literal, i.e., a string of digits containing a decimal point and which may be followed by an exponent. The notation here is the same as the notation for loading floating point numbers on "D" and "E" cards, except that in literals the decimal point must occur between two digits, and not at the beginning or end of the number. Examples: 2.0, 3.1415926535, 500.62-3, 256.15+2, etc. Illegal: 2, 2., .2, etc.

4. Any parameter algebra expression (including a single symbol or a single fixed point literal) may occur following the operation "**." Some further examples of machine algebra and display algebra follow, to clarify the above list.

MACHINE ALGEBRA	DISPLAY ALGEBRA
`Y = Z - 3.0*(Z-6.53)`	$Y = (Z-3)\ (Z-6.53)$
`AREA=R**3*4.0*3.1415926535/3.0`	$A = \frac{4}{3} \pi r^3$
`AREAT = B1 + B2*0.5*H`	$A_t = \frac{1}{2} (b_1 + b_2)h$
`DPUV = U1*V1 + (U2*V2) + (U3*V3)`	$u \cdot v = u_1 v_1 + u_2 v_2 + u_3 v_3$
`D = B + C + $`	$d = \lvert b + c \rvert$
`D = B + $ + (C + $)`	$d = \lvert b \rvert + \lvert c \rvert$
`CPUV1 = U2*V3-(U3*V2)`	$(u x v)_1 = u_2 v_3 - u_3 v_2$
`D=Z1-Y1**2+(Z2-Y2**2) + (Z3-Y3**2).$R`	$d = \sqrt{(Z_1 - y_1)^2 + (z_2 - y_2)^2 + (z_3 - y_3)^2}$
`YS1 = AX + (2.0*BX)*AX + (BX**2)` or `AX + BX**2`	$y_1^2 = a_x^2 + 2a_x b_x + b_x^2$ or $(a_x + b_x)^2$

We note that in many of the above expressions, more parentheses are needed in the machine algebra than in the display algebra counterparts. However, quite often in general equations fewer parentheses are needed in machine algebra than in display algebra. In a complete code, quite frequently the number of parentheses used will usually be less than the number needed in display algebra.

Address modifiers. Any symbol in a floating point expression may be followed by a modifier in parentheses. The purpose of these modifiers is to do one of the following:

A. To modify the address of a block in some ways, e.g., by means of parameter algebra, contents of an index register. or stored address.

B. To specify a particular arithmetic such as fixed point or

Boolean. Floating point algebra is always assumed unless one of these modifiers is used, or unless the arithmetic desired is obvious from context, e.g., by the occurrence of a fixed point literal or an index register symbol in the expression.

C. To cause an address to be interpreted in a particular way, as for instance to address the "Q" portion of a double stored number.

D. To cause only a portion of the quantity addressed to be used, such as the sign only or the magnitude only.

E. To cause all or part of the control word of a symbol to be used instead of the data addressed by the symbol.

F. To cause the contents of two locations to be swapped.

For the sake of completeness, all modifiers allowed in IVY algebra will be discussed here. Many of these are appropriate only in fixed point or Boolean algebra. A summary appears in Chapter 9, page 193. These modifiers will now be discussed in turn and the conventions illustrated by examples.

A. A symbol may be modified by parameter algebra alone, parameter algebra plus an index register, as follows (where P represents any parameter algebra, X_n represents an index register symbol, A_n represents a store address symbol) and SYMBOL (\$WA) is the control word address of the block:

ALGEBRAIC FORM	ADDRESS COMPILED
SYMBØL(P)	SYMBØL(\$WA) + P
SYMBØL(X_n + P)	SYMBØL(\$WA) + P modified by C(X_n)

ALGEBRAIC FORM (continued)	ADDRESS COMPILED
SYMBØL(X_n)	SYMBØL($WA) modified by C($X_n$)
SYMBØL(A_n)	A_n
SYMBØL($X_n + A_n$)	A_n modified by C(X_n)

*C(X_n) ("the contents of X_n") provides a dynamic address modification; the C(X_n) may be changed at will during execution of a program, thus dynamically stepping through an array or calling sequence. The C(X_n) are <u>added</u> to the specified address in order to perform this modification. On the 7090, this addition is simulated, and how this is done is of no concern here; the 7090 programmer can safely assume that his index registers add as surely as do those of the 7030. In the IVY system, index registers always contain positive values and are not allowed to assume negative values, even on the 7030, which allows signed value fields. The extra fields of the 7030 index register are not accessible to the programmer unless he uses longhand code.

The "store address" feature allows the algebraic coder the unique privilege of storing addresses, if he so desires. That is, he can first compute the address he wishes to use, assigning to it one of the "A_n" symbols, and then by placing it in parentheses as a modifier, in a later expression, cause a "store address" to insert the calculated expression. Thus the symbol modified by an "A_n" expression is a dummy; any symbol could be used, although in practice the symbol for the block addressed by "A_n" is generally used. The format for computing a stored address is as follows:

An.m = F

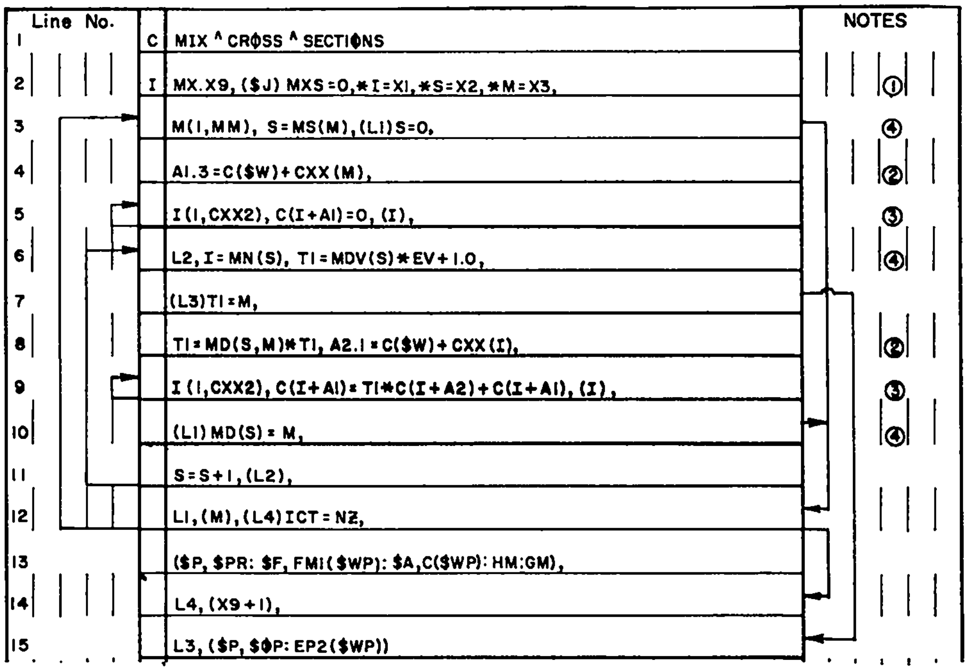
SYMBØL($X_n + A_n$) = ...

... + SYMBØL(A_n) * ...

... ("A_n" occurs "m" times)

That is, A_n is entered followed by a period and one or more digits which give the number of times "A_n" occurs in the expressions following. The letter "F" represents a fixed point expression for the address. The code which follows must contain "m" symbols modified by "A_n". Once these "m" symbols have occurred, the same "A_n" is available to be re-used to store a different address. Since only a very few distinct "A_n's" are usually necessary to encompass a given sequence of code, one can quite easily re-use each "A_n" as its field of operation is completed, and thus reduce considerably the number of "A" symbols specified on the "S" card (page 26). The number of "A" symbols specified should be minimized by adopting the above practice of re-using an "A_n" as soon as its field of action is completed. This is what is meant by "independent" store address expressions mentioned in Chapter 2, page 28. Note that "A_n" entries are "formula-limited," that is, once an "A_n" is defined, <u>all</u> symbols modified by it must appear in the <u>same</u> <u>formula</u>.

The following example should serve to illustrate the above discussion. This is a segment of an actual code, and many details in it have not yet been discussed. By the end of Chapter 5, all the techniques of this example will be clear.



Line No.	C		NOTES
1	C	MIX ^ CRØSS ^ SECTIØNS	
2	I	MX.X9, ($J) MXS=0,*I=X1,*S=X2,*M=X3,	①
3		M(1,MM), S=MS(M),(L1)S=0,	④
4		A1.3=C($W)+CXX(M),	②
5		I(1,CXX2), C(I+A1)=0, (I),	③
6		L2,I=MN(S), T1=MDV(S)*EV+1.0,	④
7		(L3)T1=M,	
8		T1=MD(S,M)*T1, A2.1=C($W)+CXX(I),	②
9		I(1,CXX2), C(I+A1)=T1*C(I+A2)+C(I+A1), (I),	③
10		(L1) MD(S)=M,	④
11		S=S+1,(L2),	
12		L1,(M),(L4)ICT=NZ,	
13		($P, $PR: $F, FM1($WP): $A,C($WP): HM:GM),	
14		L4,(X9+1),	
15		L3,($P,$ØP: EP2($WP))	

NOTES:

1. Here we see the "*" convention, which we first encountered on "D" cards (page 55), used here to define new symbols for index registers. Index registers are the only quantities for which this is allowed on "I" cards. Note also the use of the "$J" or "jump" feature, originally discussed in connection with data (page 55). If the condition following the "($J)" entry is satisfied, the formula or formula set is not assembled. No second "$J" between commas is needed. This feature is useful in case a particular formula is not used in an assembly, i.e., is not entered if the condition is satisfied. See page 104.

2. Expressions for A1 and A2 are computed.

3. Index registers plus modifiers A1 and A2 occur in these expressions.

4. Here we see modification by index registers alone.

The above example will also serve as an example for other techniques which will be described in this chapter and the next.

B. In an expression consisting entirely of symbols, and where the type of arithmetic is not obvious from context (no literals or index register symbols occur); if algebra other than floating point is desired, one of the modifiers "A" for fixed point or "B" for Boolean must be placed within parentheses, separated from the modifier of type A by a comma, after the expression to the left of the equal sign. The formats are as follows:

SYMBØL(M_A, A) = ... (fixed point expression)

SYMBØL(M_A, B) = ... (Boolean expression)

where M_A is a modifier of type A (i.e., a parameter algebra expression, index register, etc.) and "A" denotes fixed point arithmetic following the "=," and "B" denotes Boolean. Whenever one of these modifiers is present, modifier "M_A" _must_ occur. For example, if we wish to compute a value for a single fixed point quantity "CE," if the expression the right of the "=" is unambiguous, we can write

CE = ... (expression)

but if the expression is ambiguous, we must write

CE(1, A) = ... (expression)

The "A" and "B" modifiers can appear only to the left of an equal sign.

C. Modifiers for dealing with double stored numbers can occur only to the _right_ of an equal sign, and are added inside parentheses, after a comma, in the same manner as modifiers of type B. These modifiers appear in the following format, where "M_A" represents a modifier of type A, which _must_ be present, and where "P" represents any parameter algebra:

FORMAT	EXPLANATION
SYMBØL(M_A,Q.P)	"Q" portion of DS number having tag length P
SYMBØL(M_A,M.P)	Magnitude (absolute value) of "Q"
SYMBØL(M_A,T.P)	"T" portion, length P, of DS number.

Note that since the "T" portion of a double-stored number is unsigned, the magnitude ("M") modifier always unambiguously means the magnitude of the "Q" portion. Also recall that the "Q" portion may be either fixed or floating point, so that "SYMBØL(M_A,Q.P)" and "SYMBØL(M_A,M.P)" are ambiguous expressions, and if fixed point algebra is desired, the "A" modifier described above must be used left of the equal sign to specify arithmetic. However, "T" is always fixed point and defines an expression as fixed point unless other arithmetic is specified. Examples of these modifiers follow. In the section on fixed point algebra we will encounter further examples:

EXAMPLES	NOTES
SN(X1+3) = AGT(X2,T.3).$CA*FN(X2)/3.15621-06	1
AF(X3+GE+17) = RST(1,Q.5)//CX(N1+3) - CRYZ(2,Q.7)	2

NOTES: 1. In this example the use of "T" in a floating point expression is permitted, since the quantity is followed by the operation ".$CA" which converts it to a floating point number.

2. The "Q" portion of two numbers having different tag lengths are used in this algebra.

If it is desired to <u>compute</u> the "Q" or "T" portion of the quantity to the left of the equal sign, one must use <u>expression</u> modifiers, described below on pages 83-84.

D. Modifiers of type D are used to impose the sign of a quantity on the result of the previous calculation, or to ensure that only the magnitude (absolute value) of a quantity takes part in an operation, and in one case, to save the remainder of a division or the low-order part of any floating point operation, for double precision purposes. Type D modifiers can occur <u>only</u> to the <u>right</u> of an equal sign. These modifiers are as follows:

FORMAT	EXPLANATION
SYMBØL(M_A,M)	use magnitude of addressed quantity
+SYMBØL(M_A,$̸)	impose sign of addressed quantity on previous result
-SYMBØL(M_A,$̸)	impose negative of sign of addressed quantity on previous result
*SYMBØL(M_A,$̸)	multiply sign of addressed quantity by sign of previous result
SYMBØL(M_A,R)	save the low order part of the result of this operation in the IVY location "$̸CS1"

An example of the magnitude modifier "M" is shown in line 8 of the coding example on page 78. Other examples:

EXAMPLE	NOTE
RXN(X1+A2) = GPG(X3)**3+GE(1,$̸)	1
APG = ALPHA(X3+N) + B(X2) *STV(X1+A1,$̸)	2
SUMY = AB(X1+1,R) + AC(X1+1) + AD(X1+1)	3
EMG(X5+3+GE) = SRN(X2+1,R) -SRT(X2+1) + $̸/FNT3	4

NOTES: 1. The sign of GE1 is attached to the result of the previous operation.

2. The signs of STV(X1+A1) and the preceding result are multiplied, and this resulting sign is attached to that result. If the signs are alike, "+" will result; if unlike, "-" will result.

3. The low order part of the floating point result of this calculation is stored in "$CS1," from which it may be obtained for double precision work. We recall that on the 7090, both a high- and low-order part are carried in all floating point operations; on the 7030, the "R" is a signal to execute double precision operations followed by a "store low order" instruction to "$CS1." The "R" modifier, if at all possible, should always occur as near to the beginning of the expression as possible in order to speed compilation.

4. In this case, the remainder of the division, if any, is stored in "$CS1" once the operation (division) has been completed.

E. Address modifiers of type E cause the symbol to be interpreted so that the operand becomes all or part of the control word, or the address of the control word, associated with that symbol. Ordinarily these modifiers are used <u>only</u> with fixed point and index arithmetic. All of these modifiers except "$WP" are allowed either to the left or to the right of an equal sign. Type E modifiers are as follows:

MODIFIER FORMAT	OPERAND GIVEN
SYMBØL($W)	control word
SYMBØL($WA)	control word address
SYMBØL($WC)	control word count
SYMBØL($WP)	position of control word

Note that no other modifiers of any type may appear in parentheses with the control word modifiers. The chief use of control word modifiers is in the computation of stored addresses (see lines 4 and 8 of the example on page 78), for which "$W" is ordinarily used; the computation of index register values, using "$WA," "$WC," and "$WP"; and in manipulations involving the symbol table. Examples of these latter two uses are found in the sections of this chapter dealing with index and fixed point algebra, and in Appendix 1.

F. The swap modifier "S" always appears to the <u>right</u> of an equal sign and must follow a type A modifier. The format is as follows:

$$\mathrm{SYMB\O L_1(M_A) = SYMB\O L_2(M_A,S)}$$

The contents of the two locations are simply swapped, i.e., interchanged. No arithmetic is permitted to the right of the equal sign.

<u>Expression modifiers</u>. Two modifiers, known as expression modifiers, may be appended to the end of an expression in order to specify that the result is to be stored in the "Q" or "T" portion of the quantity which appears to the left of the equal sign. These modifiers appear as follows, where "M_A" represents a modifier of type A which may or may not be present, and where "P" represents any parameter algebra:

$$\mathrm{SYMB\O L(M_A) = (expression).\$Q.P}$$

$$\mathrm{SYMBOL(M_A) = (expression).\$T.P}$$

Ordinarily the modifier ".$T.P" should follow only fixed point expressions. ".$Q.P" may follow either fixed or floating point expressions, since the "Q" portion of a double-stored number may be either fixed or floating point.

Special symbols addressable by IVY algebra. Four of the special "$" symbols in IVY represent data blocks and may be addressed by algebraic code. All of these except "$M" may be modified by modifiers of all types. However, the control word modifiers have a somewhat different meaning when attached to "$CS" and "$Z"; this is covered in Chapter 5 in the section on calling sequences, pages 108-110. These special symbols and their meanings and usage are as follows:

1. $M. This symbol may occur only to the right of an equal sign, without modifiers. It means, "repeat the quantity to the left of the equal sign." For example, the following two expressions are equivalent:

```
AD(X2+GE*3) = AD(X2+GE*3)*SN(X3+2)/FN5
```

and

```
AD(X2+GE*3) = $M*SN(X3+2)/FN5
```

2. $CS. This symbol represents the "calling sequence data block" and may appear on either side of an equal sign, with or without modifiers. Generally "$CS" is used to convey information to, or to receive information from, a subroutine. We have already encountered another use of this block: the low order part of a double precision result is stored in the location "$CS1." The "$CS" block is twenty words long and can be used the same as any data block except that it should be recalled that the contents of "$CS" are destroyed by some

subroutines. Further discussions of "$CS" are found in Chapter 6, pages 127-129.

3. $Z. This symbol, which must always have at least a modifier of type A, may occur on either side of an equal sign. It simply means "supply an address of zero." Its chief use is in store address expressions of the form

$$\$Z\ (X_n + A_n,\ M_2)$$

where the symbol is unimportant, since "A_n" is the address actually used, and in subroutines, to refer to entries in a calling sequence, in the form

$$\$Z\ (X_n + N,\ M_2).$$

("M_2" represents either a null field or some type of legal modifier other than type A.) The usage of "$Z" for the latter purpose is discussed in Chapter 5, pages 108-110.

4. $D, $DA, $DB,...,$DZ. These 27 special symbols are shared by subroutines and are used for internal data. A complete description of their usage is given in Chapter 5, pages 110-112.

5. $L. The "$L" symbols provide access to certain special constants and addresses used by IVY. The symbols address the following information: $L1 = FAC (first address for code); $L2 = FAD (first address for data); $L3 = NLA (next loading address for code); $L4 = NBA (next block address for data); $L5 = 7090_{10} if machine is 7090, 7030_{10} if 7030; $L6 = number of remark characters per word. $L1, $L2, $L5 and $L6 are

available for testing purposes only. $L3 and $L4 may be altered with discretion, as described in Appendix 1, pages 203-209.

Statement separation and continuation. As one can observe from examining the example on page 78, IVY statements in the algebraic language are separated by commas. There may be any number of statements on a card, of up to 71 characters in length. An algebraic statement can be continued from one card to the next provided that symbols, literals, symbol modifiers in parentheses, expression modifiers, and operations of more than one character ("**," "+$," etc.) are complete on one card. These items which cannot be split from one card to the next are called units of an expression; thus we can say that expressions can be continued from one card to the next provided that units of the expression are complete on one card.

*Blanks occurring in expressions are always ignored, as we can see by again referring to the example. Thus blanks may be used, if desired, to separate units of the expression for easier reading. The carat"^" is used to denote blank spaces on the coding sheet; if more than one blank is desired, the notation is to write the number "n" of blanks enclosed in a box, thus: $\boxed{n}$. Note that one need not use these conventions to represent blanks occurring at the end of the card. In general, blanks are totally ignored on every type of IVY card except the remark card where they form part of the input data.

Fixed point algebra. The same operations are allowed in fixed point algebra as in floating point algebra, with some changes in meaning caused by the peculiar nature of fixed point algebra. These differences are as follows:

OPERATION	MEANING
/	divide, and truncate quotient to integer.
//	reciprocal divide, and truncate quotient to integer.
.$CX	convert fixed point number to exponent of floating point number, i.e., give $2^{(\text{result})}$ in floating point.
.$CA	convert fixed point number to floating point number.
.$R	take square root, and truncate result to integer.

These differences are, of course, occasioned by the difference between floating and fixed point arithmetic. Fixed point arithmetic is the arithmetic of integers; hence the difference in the divide instructions.

The same operands, with the exception of literals, and with the addition of symbols for "K" blocks (page 182), are allowed in fixed point algebra as in floating point. Literals, of course, must be fixed point decimal numbers, i.e., a string of digits not containing a decimal point. In addition, symbols for index registers are allowed in fixed point algebra; when these symbols are used, the contents of the index register are used as an operand. When an index register appears in an expression, there are two modes of operation: immediate and direct. The "direct" mode is signalled by the modifier "A." When the "A" is missing, immediate algebra is assumed; that is, the expression is assumed to be parameter algebra and is computed according to the values of the parameters loaded at the start of the deck.

The same address and expression modifiers are allowed in fixed point algebra as in floating point algebra. The same conventions are also used for parentheses, the continuation of statements from one card to the next, and the use of special "$" symbols.

The following examples of, and notes on, fixed point algebra should serve to illustrate all necessary conventions.

EXAMPLE	NOTE
ARX(X1+3,A) = VDBC(X3,M) + RX3/AGT5*$	1
AD(1,A) = X3*VXC1 + PAR2	2
AE = X5 + GE*TH/3	3
FRN(X2) = 3 + ART(X1)*56-SRTN(X2,$).$CA.$Q.5	4
A3.6 = AD($W) + ADX(X1) + ADXX(X2)	5
INDEX = X1 - VRN($WC)	6

NOTES:

1. We note that in this example the "A" modifier is used to specify fixed point arithmetic, since the expression, containing only symbols, is ambiguous. Also note the use of the "magnitude" modifier and of "*$" to change the sign of the expression.

2. In this second example the "A" modifier is used in a different sense, since the expression contains a symbol for an index register and hence is unambiguous. The "A" is a signal that the algebra is "direct" or "dynamic," i.e., the computed values of the symbols at the time of <u>execution</u> indicated are used in the algebra.

3. Here again the expression involves the contents of an index register, but since the "A" is not specified, the arithmetic is assumed to be "immediate" or "static," i.e., the values of the specified parameters at the time of <u>assembly</u> are used in the algebra. This is done by placing the operands directly

into machine instructions, by using immediate arithmetic on the 7030 and immediate-type instructions such as "TXI" on the 7090.

4. In this fixed point expression the "A" is not needed since the occurrence of fixed point literals makes it unambiguous. Note that the expression is converted to floating point and then stored in the "Q" portion of the double-stored block "FRN," having tag length 5.

5. This is an example of the computation of a stored address. Recall that other examples of this were shown in the coding example given on page 78. The usual expression for a stored address includes the control word of a block (from which the base address is obtained) modified by the addition of one or more index multiples under the control of index registers. The philosophy of this technique is discussed in detail in Chapter 8, pages 168-169.

6. This is an example of an immediate indexing operation in which the control word count is subtracted from the contents of the index register.

Index register algebra. Although index registers, as we have seen, can appear in fixed point algebra, true index register algebra differs considerably from fixed point algebra. Index register algebra is denoted by the occurrence of an index register symbol to the left of the equal sign. The operation set for index register algebra is as follows:

OPERATION	MEANING
+	add
-	subtract
.$\not{\$}$X	if previous result is negative, set to 1

Thus we see that the operation set for index arithmetic is quite restricted. However, under most circumstances the operation, "*," for instance, is not needed since tables of index multiples can be constructed on "D" cards and used in index arithmetic; and constructing index multiples (done automatically by IVY) is the chief reason for the existence of a "multiply" operation. If it is desired to load an index from a more complicated expression, one can first use fixed point algebra to compute the expression, and then load the index from the location where the result was stored.

It has been remarked before that index register contents are restricted to positive, non-zero values. The purpose of the ".$X" instruction is to keep the index register contents positive by guaranteeing that if the result of an expression is negative, a positive result of 1 will be substituted. The magnitude of index register expressions must be less than 2^{15} on the 7090 and 2^{18} on the 7030. Note that if the result exceeds these bounds, the number given will be truncated modulo 2^{15} or 2^{18} as the case may be.

One modifier is allowed left of the equal sign in index register expressions: "A" separated from the index register symbol by a period. The purpose of this "A" is the same as in fixed point algebra when index registers are present, to specify "direct" arithmetic. Index register algebra is always "immediate" if the "A" is not present. Of course, here as elsewhere, the apellations "direct" and "immediate" apply only to units of the expression other than index register symbols. In either fixed point or index algebra, the contents of the index register at execution time form the operand.

Some examples of index register algebra follow:

EXAMPLE	NOTE
X1 = X1 + 1	1
X3.A = AD + 3	2
X3 = GE + 3	3
X2.A = AE-INDEX.$X	4

NOTES:

1. The contents of index 1 are incremented by 1.

2. In this case, direct algebra is specified by the "A." The contents of "AD" at execution time are incremented by 3 and placed in index register 3. Note that "AD" is the symbol for a rather complicated expression illustrated in the section on fixed point algebra. This example thus shows how a dynamic loading of an index register can be performed.

3. Immediate algebra is assumed here, which means that the contents of GE at compiling time, plus 3, are placed in index register 2.

4. Here the arithmetic is performed in the direct or dynamic sense, and the index register contents are set to 1 if the result is negative.

Renaming of an index register. An index register may be renamed at any point in the code by the use of the "*" convention which was originally discussed in connection with data blocks on page 55. Usually an index register will be renamed to a single-letter symbol to save the necessity of writing the two or more characters associated with every "X_n" symbol. If it is desired to use a symbol of two letters or more to rename an index register, the symbol should of course have been defined by its appearance on a "D" card. The format for renaming an index register is as follows:

$$*SYMBØL = X_n,$$

where "SYMBØL" represents any single letter symbol (except A, X, or L), or any symbol of more than one letter which has been previously defined on a "D" card, i.e., entered in the symbol table.

Using what we now know, let us construct a simple example of an index loop. An index loop is specified by placing the values between which the index is to run, separated by a comma, in parentheses after the name of the index register, at the beginning of a loop. The end of the loop is denoted by placing the index register symbol in parentheses. We have encountered other examples of index loops in the example on page 78, lines 5, 9, and 3-12. The following loop is for the simple purpose of constructing the dot product "D" of two vectors "VA" and "VB," each having three components:

```
I|*I = X1, D = 0,I(1,3), D = D + VA(I)*VB(I),(I),...
```

or, equivalently,

```
I|*I = X1, D = 0,I(3,1), D = D + VB(I)*VA(I),(I),...
```

At least one of the limits of an index loop must be 1. The other may be represented by a literal, as above, or by a symbol, as in the example on page 78, or by a parameter algebra expression. The operation is performed for the first value of the index, and then the loop is reiterated after the index has been increased or decreased by 1, until the index reaches the final value. If it is desired to construct a loop for which one of the limits is not 1, or for which the index increment is not 1, or where the index is to run between computed (as opposed to parameter) values, other techniques must be used, utilizing the "L_n" entry. Examples of this appear in Chapter 5, page 99 , and Chapter 8, pages 169-171.

Boolean algebra. Boolean algebra is used for performing logical operations by obtaining a result involving a bit-by-bit comparison of two or more operands. The set of Boolean operations is as follows:

OPERATION	NOTE	MEANING
+	1	logical add, sometimes called "inclusive 'or'"
*	2	logical multiply, sometimes called "and"
'	3	take one's complement of preceding
.$U	4	give 1 if result is $\neq 0$
.$V	4	give 0 if result $\neq 0$, otherwise give 1.

NOTES:

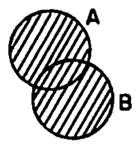
1. The inclusive "or" of two binary numbers is obtained by comparing the numbers bit-by-bit, and setting the corresponding bit of the result to 1 if either or both operand bits are 1, and to zero otherwise. For example, the inclusive "or" of 101101011101 and 001011100101 is 101111111101.

2. The "and" of two binary numbers is obtained by comparing the numbers bit-by-bit, and setting the corresponding bit of the result to 1 if both bits are 1, and to zero otherwise. The "and" of the two numbers given above is 001001000101. Note that the Boolean sum of the exclusive "or" and the "and" is the inclusive "or."

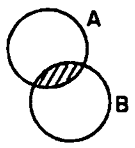
3. The 1's complement of a binary number is obtained by replacing all 1's with zeros, and all zeros with 1's. For instance, the 1's complements of the two numbers in note 1 are 010010100010 and 110100011010.

4. These two operations are the same as the corresponding ones in fixed and floating point.

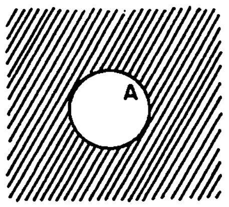
One can represent the Boolean operations graphically, as is illustrated below, assuming we have two intersecting regions "A" and "B." The result of the operation is the shaded area. We might say that the area common to both regions corresponds to the bits of both binary numbers which are equal to 1, and the rest of the area corresponds to the differing bits of the binary number. The area outside the regions corresponds to the bits of both numbers which are zero.



A+B



A*B



A'

From these illustrations a few identities of Boolean algebra become evident. For instance, the <u>exclusive</u> "or" of two numbers is equal to the logical sum of the "and" of the first number and the complement of the second, and the "and" of the second number with the complement of the first. That is, the following expression produces the exclusive "or":

Y(1,B) = V*(W') + (W*(V'))

The <u>exclusive</u> "or" of two binary numbers is obtained by comparing the two numbers bit-by-bit, and setting corresponding bit of the result to 1 if the two bits differ, and to zero otherwise. If we use the two numbers of note 1, page 93, this can be verified for a particular case:

101101011101 (exclusive "or") 001011100101 = 100110111000

and

101101011101*(0010111001011) + (001011100101*(1011010111011)) =

101101011101*110100011010 + (001011100101*010010100010) =

100100011000 + 000011000000 = 100110111000.

In this case, both results are the same. In general this is true, although no proof is offered here. Many other similarly interesting relationships between Boolean operations can be discovered by studying the diagrams. It is possible to obtain sixteen possible results by combining two numbers using the IVY set of Boolean operations; these sixteen results make up the entire set of sixteen so-called <u>logical</u> <u>connectives</u>.

The following further observations apply to the Boolean set:

1. The operations "+" and "*" are commutative and associative, i.e., B + C = C + B, B*C = C*B, and D*(B*C) = (D*B)*C, etc.

2. The operation "*" is distributive over "+," and "+" is distributive over "*," i.e.,

   D + B*C = D*C+(B*C);

   D*B + C = D + C*(B + C).

3. The "and" (logical product) of two numbers contains less one bits than either number unless both numbers are equal;

the inclusive "or" (logical sum) of two numbers contains more one bits than either number unless both numbers are equal. The "and" and inclusive "or" of equal numbers are equal to the two numbers; the exclusive "or" of equal numbers in zero.

Boolean expressions must always be denoted by the "B" modifier to the left of the equal sign. The algebra to the right of the equal sign may contain symbols for Boolean blocks or Boolean literals. Ordinarily Boolean expressions should not contain symbols for non-Boolean quantities unless great care is exercised. While some very useful computations can be carried out by violating this rule, such computations usually will not work on all machines for which IVY is available, since the formats of internal words differ. For example, on the 7090, the following two expressions are equivalent and would compile the same sequence of instructions:

```
ADF(1,A) = AXCG(3+X3) .$CX
```

and

```
ADF(1,B) = AXCG(3+X3)*377000000000,ADF(1,A) = ADF/(2**27) - 129,*
```

whereas on the 7030, the latter expression will definitely not do the same as the former, because of the differing word lengths and floating point formats on the two machines.

*In the last expression, "$M" can be used instead of "ADF" on the right of the equal sign, if desired.

CHAPTER 5

FLOW OF CONTROL, CALLING SEQUENCES, AND THE EXECUTE STATEMENT

L-entries. "L-entry" is the term applied to the use of a numbered "Ln" symbol for branching purposes. An "L-entry" may be used for both conditional and unconditional branching. In the algebraic language the entry point is marked by the occurrence of an "Ln" symbol between commas (for longhand conventions see Appendices 2 and 3). Unconditional branching to the statement immediately following this entry point is specified by the occurrence of the same "Ln" symbol in parentheses between commas. This branching may be performed in either a forward or backward direction, thus:

```
    ...,(Ln),...(algebra)...   →┐
                                │
    .....,Ln,....(algebra)...  ←┘
                               ←┐    flow of control
                                │
    .....,(Ln),...(algebra)... →┘
```

An entry of "Ln" between commas for a particular value of "n" can occur only once in a given formula. Conditional branching to a given "Ln" entry is specified by the entry of "Ln" in parentheses, followed by a modifier, if necessary, specifying the type of algebra used in the expression to the right of the right parenthesis. If the given condition is satisfied, the branch is performed. Otherwise, control proceeds to the next algebraic expression. The general format is as follows:

,(Ln,M)Algebra = C,

where

1. "Ln" represents the entry to which branching is to be performed.

2. "M" represents one of the modifiers "A" (for fixed point), or "B" (for Boolean), if necessary to specify the type of algebra to be performed in the following expression. "M" and the comma preceding it may be omitted if the algebra is unambiguous according to the tenets of Chapter 4.

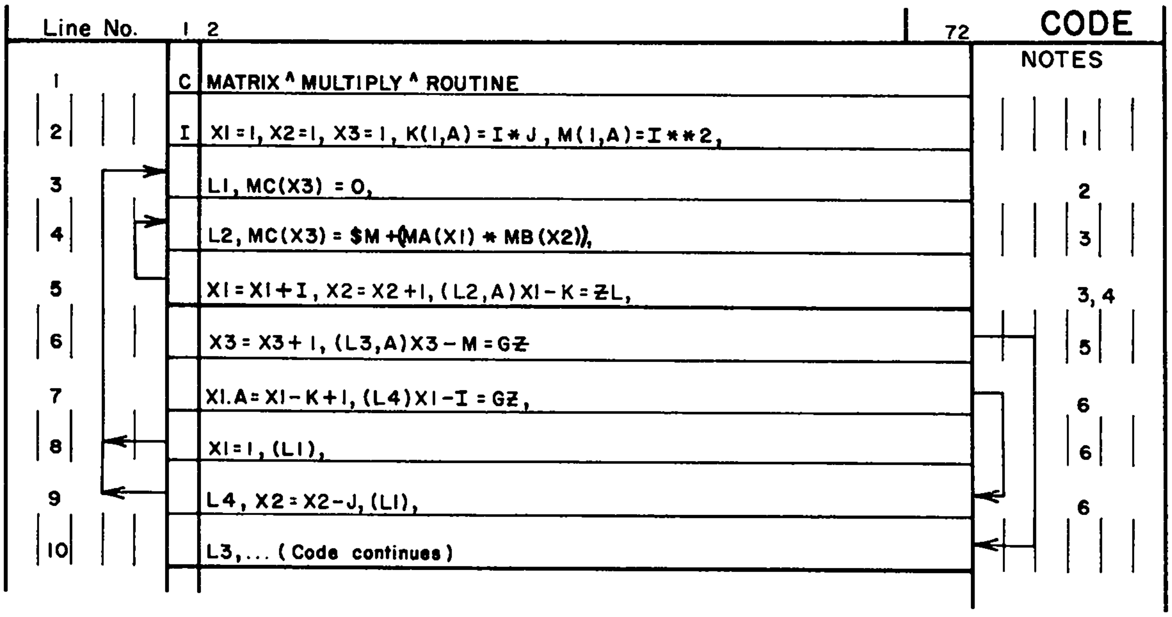
3. "Algebra" represents any machine algebra expression.

4. "C" represents one of the following conditions:

C	CONDITION
Z or 0	branch if result is zero
NZ	branch if result is not zero
GZ	branch if result is greater than zero
LZ	branch if result is less than zero
ZG	branch if result is zero or greater
ZL	branch if result is zero or less
P	branch if result is plus
M	branch if result is minus

Examples. An examination of the coding example in Chapter 4, page 78, will reveal an unconditional branch on line 11, and conditional branches on lines 3, 7, 10, and 12. The "Ln" entries to which branching is performed are on lines 6, 12, 14, and 15. Note how the flow of control is marked by arrows.

The following example also makes use of conditional and unconditional "Ln" branching. This example performs the matrix multiplication of the I × J matrix "MA" times the J × I matrix "MB" and stores the I × I result into "MC."



Line No.	1	2 … 72	CODE NOTES
1	C	MATRIX^MULTIPLY^ROUTINE	
2	I	X1 = 1, X2 = 1, X3 = 1, K(1,A) = I * J , M(1,A) = I ** 2,	1
3		L1, MC(X3) = 0,	2
4		L2, MC(X3) = $M + (MA(X1) * MB (X2)),	3
5		X1 = X1 + I , X2 = X2 + 1, (L2,A) X1 - K = ƵL,	3, 4
6		X3 = X3 + 1, (L3,A) X3 - M = GƵ	5
7		X1.A = X1 - K + 1, (L4) X1 - I = GƵ,	6
8		X1 = 1, (L1),	6
9		L4, X2 = X2 - J, (L1),	6
10		L3,... (Code continues)	

NOTES:

1. All three index registers used in the code are initialized to 1. The quantities "K" and "M" are computed for later use in index comparison, since, as we recall, the operations "*" and "**" are not allowed in index algebra. The "A" modifier is necessary in both since the algebra is ambiguous. There are no literals present to distinguish the expressions from floating point.

2. The current element of matrix "MC" is initialized to zero before computation begins, since the result is computed in a cumulative fashion.

3. The current element of matrix "MC" is increased by the products of the appropriate elements of "MA" and "MB." Then the "MA" index, X1, is increased by I, and the "MB" index, X2, by 1, since we are proceeding through a row of "MA" and a column of "MB." Note that the increment of X1 is by a parameter, and X2 by a literal; hence both operations are immediate and the ".A" modifier is not needed.

4. We proceed back to increment "MC" again if the row of "MA" is not yet exhausted. Note that the ".A" modifier is used here to specify direct, or dynamic, index algebra in the test of X1 since "K" is a computed quantity.

5. If the row of "MA" is exhausted, the index for "MC" is incremented by 1. (We are computing "MC" column-wise). If "MC" is exhausted, exit is made to "L3." Note again the use of ".A" to specify dynamic index algebra.

6. X1 is now incremented backwards so that "MA(X1)" will start the next row in "MA." If "MA" is exhausted, we start over at its beginning but proceed to the next column in "MB." If "MA" is not exhausted, we proceed to its next row, but use the same column of "MB." Note that the ".A" modifier is not used when I and J are involved, since these quantities are parameters, being array dimensions, and hence can be used in immediate arithmetic.

It should be noted that the above example is included for illustrative purposes only, and is not intended to demonstrate the best possible technique for multiplying two matrices. The method illustrated is used only because it is a familiar one.

Restrictions on L-entries. As has been mentioned before, L-entries are purely local entities within a formula. Branching from one formula to another by means of L-entries is prohibited. An L-entry in a formula to which no branch is performed, and a branch to a non-existent L-entry are detected as errors, and a diagnostic printout is performed. If execution reaches the point where a branch to a non-existent L-entry occurs, and the branch is successful, IVY takes control, prints a comment to the effect that execution cannot proceed beyond this point, and selects the card reader or off-line tape in an attempt to process the next job, if any.

Pathfinder branching. Pathfinder branching is a means of controlled entry to subroutines, when a subroutine is entered from several points in the code and return must be made to whichever point that performed the entry. This is done by using the pathfinder register, called "$P." When a pathfinder branch is made, the location from which the branch occurs is stored in the pathfinder. The subroutine, then, can load the contents of the pathfinder into some available index register and return by branching using this index register. The format for a pathfinder branch to an L-entry subroutine is as follows:

```
,($P,Ln),
```

Note that the pathfinder branch is unconditional. The subroutine named "Ln" begins with the entry:

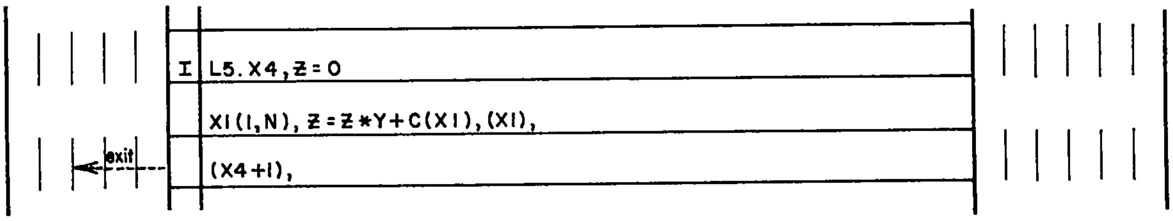
```
,Ln.Xm,
```

where "Xm" is the symbol for the index register into which the pathfinder contents are loaded. The subroutine then ends with the entry

```
,(Xm+1),
```

where "Xm" is the same index register which was loaded with the pathfinder contents. This entry causes control to return to the expression following the pathfinder branch to the subroutine.

<u>Examples of a pathfinder branch and subroutine</u>. Suppose that at several points in a formula we calculate a quantity "Y" and we wish to evaluate a polynomial "Z" of Nth degree for this "Y." Assume the coefficients of the polynomial are stored in order of decreasing powers of the variable in the block "C." If the subroutine is called "L5," it might look as follows:



```
I  L5.X4,Z=0
   XI(I,N),Z=Z*Y+C(XI),(XI),
   (X4+I),
```

The contents of the pathfinder are placed in X4. The polynomial is then set to zero initially, and then each time through the loop, it is multiplied by "Y" and increased by the next coefficient. After the evaluation loop is completed, the exit is performed to (X4+1). Note also that the exit from a subroutine must be unconditional.

<u>Formulas and formula branching</u>. As has been remarked previously, a formula is a subset of a formula set. The name of a formula is defined solely by its appearance on an "I" card and must not have been defined on a "D" card. The format for entry of a formula name is the same as for an L-entry, namely,

$$,\text{FØRM}, \text{ or, } \text{FØRM}.X_n,$$

depending on whether or not the formula is entered by a pathfinder branch. "FØRM" represents any legal symbol not previously defined, and

"X_n" any free index register. Similarly, branches to a formula entry may take any of the following forms:

BRANCH	TYPE
,(FØRM),	unconditional
,(FØRM,M)Algebra = C,	conditional
,($P,FØRM),	pathfinder

A formula may be a subroutine accessible to any formula within the same formula set, just as an L-entry can be a subroutine within any formula. An example of a formula which is a subroutine is the formula "MX" of the example in Chapter 4, page 78. This formula is entered by the instruction

,($P,MX),

and we see that the contents of the pathfinder are placed in X9, and return is made by the usual entry

,(X9+1),

Formula sets. The name of a formula set is distinguished from the name of a formula by the fact that it has been defined on a "D" card. A formula set consists of one or more formulas. In addition to its collection of formulas, a formula set may contain a short code controlling entry to its various formulas, as well as one or more branches to other formula sets. The format of a formula set entry and branches to formula sets is the same as for formulas, namely:

ENTRY	BRANCH	TYPE
,FS,	,(FS,M)Algebra = C,	conditional
,FS.Xn,	,(FS),	unconditional
	,($\not{s}$P,FS),	pathfinder

There are no restrictions on branches to formula sets. These branches may be forwards or backwards, and any formula set may contain a branch to any other formula set. Of course, the formula set to which the branch is performed must be converted and in core for the branch to be legal; otherwise, if a successful branch to a non-existent formula set is encountered, IVY regains control, prints out a comment, and begins searching for the next job.

The "jump" feature for formulas and formula sets. We first encountered the "jump" feature in connection with the definition of data, page 55. The "MX" code in Chapter 4, page 78, illustrates this feature in connection with formulas and formula sets. Using this convention, one can skip the assembly of any formula or formula set which is not entered in a particular run, under parameter control. The format for this entry is as follows:

I|F.Xn,($\not{s}$J)P = C,

where "P" represents parameter algebra and "C" represents one of the familiar conditions. If the condition is satisfied, the assembly program "jumps" to the next formula or formula set without assembling the current one. Unlike the use of "$J" with data, a second "$J" is not needed in code, since the assembly program detects the end of the "jump" field by detecting the name of the next formula or formula set as the case may be.

Calling sequences in code. Any pathfinder branch to an L-entry, formula, or formula set may contain a calling sequence. The calling sequence consists of items of information separated from each other by colons, each entry between colons representing a full word. These entries are described in detail in the treatment of the "K" card, Chapter 3, page 64. Calling sequence word entries follow the pathfinder branch in parentheses, separated from it by a colon in the following format:

```
,($P,SUBR: CS1: CS2: CS3:...: CSN),
```

where "CS1," "CS2," etc., represent calling sequence word entries. The symbols in calling sequence word entries can refer to data, remarks, or calling sequence blocks defined by "K" cards. IVY internal symbols starting with"$" cannot appear in calling sequence word entries because of the near impossibility of distinguishing between them and the "$XXX" entry. A description of addressing conventions for calling sequence words follows the next section.

Returns to a calling sequence. If a calling sequence on instruction cards contains N calling sequence word entries, return is made to the expression following the calling sequence by the branch

$$,(X_n + N + 1),$$

where "X_n" is the index register which has been loaded with the pathfinder contents. Thus, if a pathfinder branch is not followed by calling sequence word entries, the return is effected by

$$,(X_n + 1),$$

as shown earlier in this chapter.

Sometimes a subroutine may have more than one exit, the simplest case being when there is an error exit and a normal exit. Any time a subroutine has, say, M exits, the M-1 expressions after the calling-sequence-pathfinder branch expression must be simple pathfinder branches to routines which handle these extra cases. This is to ensure that the <u>extra returns comprise full words in all cases</u>. On the 7030, pathfinder branches comprise full words, while most other branches do not. For example, suppose "MATINV" is a matrix inversion routine which has two exits, the first an error exit if the matrix is singular, and the second, a normal return where the inverse has been computed. The calling sequence would then appear as follows:

```
($P,MATINV: AD($W): AE($W)), ($P,ERR),...(computation proceeds)
```

where "AD" is the square matrix of which the inverse is desired, and "AE" is the block where the inverse is to be stored. "ERR" is a routine which handles the erroneous case when the matrix is singular. The error return is marked by the full-word pathfinder branch to "ERR." The subroutine exits to the error return by the entry

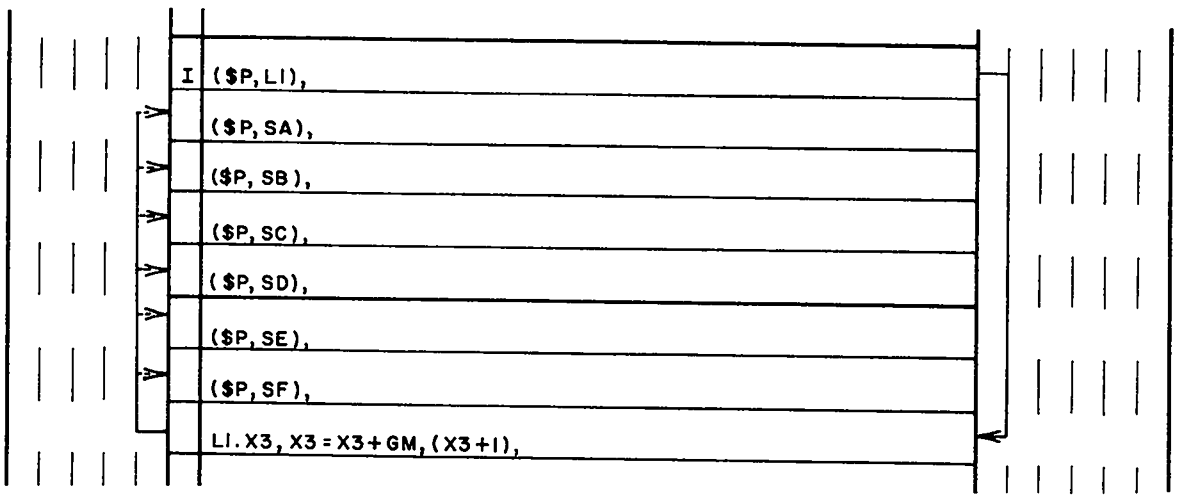
$$(X_n + 3),$$

and to the normal return by the entry

$$(X_n + 4),$$

where "X_n" is the index register containing the pathfinder contents.

Another example of a "calling sequence" with several returns is the <u>transfer table</u>. A transfer table is essentially a calling sequence made up entirely of pathfinder branches, each representing a branch to an alternative subroutine. Which entry of the transfer table is used depends on the value of some quantity, either a parameter or a computed value. The following example illustrates the usage of a transfer table:



```
I ($P,LI),
  ($P,SA),
  ($P,SB),
  ($P,SC),
  ($P,SD),
  ($P,SE),
  ($P,SF),
  LI.X3,X3=X3+GM,(X3+1),
```

"GM" may take on one of the values 0, 1, 2, 3, 4, or 5, and depending on this value, one of the pathfinder branches to the routines "SA," "SB," etc., is executed. In this case, as in every case where a subroutine has more than one exit, the various alternative returns contain pathfinder branches.

Addressing calling sequence word entries from within the subroutine using them is done by using the symbol "$Z" modified by the pathfinder index register and a parameter algebra expression. An additional modifier expression may be separated from the first by a comma, to make it possible to extract either the "$WC" or "$WA" portion of a calling sequence word. For example, consider the matrix inversion routine whose calling sequence was given in the previous section:

($P,MATINV: AD($W): AE($W)),($P,ERR),...

In this subroutine it will be necessary to have both the count and the base addresses of these matrices in index registers in order to proceed. We shall also need the square root of the count of the matrices to show the row and column size. This can be done by the following sequence of instructions:

I	MATINV. X5, X1 = $Z(X5+1, $WC),	1
	X2 = $Z (X5+1, $WA), X3 = $Z(X5+2, $WA),	2
	X4 = $Z(X5+1, $WC). $CA. $R. $CA,...	3

NOTES:

1. The pathfinder contents are placed in X5; then the count of the first calling sequence word, i.e., the count of block "AD," is placed in X1. We do not worry about the count of "AE" since presumably it is the same.

2. The base addresses of the blocks "AD" and "AE" are placed in X2 and X3, respectively. Note that although the "$WA" modifier can be used, its appearance is not necessary, since normally an index register is loaded from the "$WA"portion of a calling sequence word, unless overruled by "$WC."

3. The count of the block "AD" is converted to floating point, its square root is taken, and it is then converted back to fixed point and placed in X4.

Actually, a great deal of work can be saved by constructing the calling sequence to "MATINV" in a more efficient manner. The above example was included only for illustrative purposes; a better way of constructing the calling sequence and the first few orders of "MATINV" is as follows:

```
I  ($P, MATINV: AD($WC): AD($WA): AE($WA): I ), ($P, ERR),
   :
I  MATINV. X5, X1 = $Z(X5+1),
   X2 = $Z(X5+2), X3 = $Z(X5+3),
   X4 = $Z(X5+4),...
```

The same index registers are loaded with the same quantities ("I" is the row or column dimension of "AD"); the only difference now will be that the exit from MATINV to the error return is:

$$(X5 + 5),$$

and to the normal return:

$$(X5 + 6),$$

In general it is good practice to make calling sequence words as simple as possible, even if more space is consumed, since the manipulations may be greatly simplified and therefore a less probable source of error.

Subroutines with variable-length calling sequences can also be constructed by the programmer, either by placing the calling sequences on

"K" cards or, with more care, placing them (as above) on "I" cards. Some further examples of this are contained in Chapter 8, pages 176-183, and will not concern us here.

Usage of the "$D" blocks. The twenty-seven "$D" symbols ($D, $DA,$DB,...,$DY,$DZ) permit different formula sets to share the same set of symbols, which, however, can unambiguously be assigned different values, addresses, and lengths in each formula set in which they are used. Generally "$D" symbols are specified in formula set subroutines which are to be used by several different programs, in order to avoid using regular alphabetic symbols and thus running the risk of having symbols which may conflict with some of the symbols used by a particular code containing the subroutine.

One formula set cannot reference, either directly or indirectly, the "$D" blocks of another formula set. Thus a subroutine can safely use its "$D" blocks for necessary constants and/or temporary storage, without the fears that the constants will be destroyed by another routine and that its own references to these blocks will destroy another routine's data.

"$D" symbols may participate in algebraic expressions like alphabetic symbols, with the following exceptions:

1. The modifiers "$W" and "$WP" are not allowed following a "$D" symbol.
2. The modifiers "$WA" and "$WC" are allowed only in immediate index arithmetic expressions involving the "$D" symbol.
3. "$D" symbols must never occur in calling sequences;

they cannot be distinguished from the "SXXX" calling sequence entry.

4. "$\not{\$}$D" symbols may not appear in parameter algebra expressions.

"$\not{\$}$D" blocks are defined on "I" or "L" cards much the same way as data blocks are defined on "D" cards, but with considerably more restrictions. Definition of "$\not{\$}$D" blocks on "L" cards is discussed in Appendices 2 and 3 in connection with longhand instructions. On "I" cards, the definition of "$\not{\$}$D" blocks <u>must</u> occur immediately after the entry of the name of the formula set in which they are used, as follows:

```
I|FSNAME.Xn,..."¢D" definitions ...,
 |... Code continues ...
```

Thus all "$\not{\$}$D" symbols used in a formula set must be defined between the entry of the formula set name and the first algebraic expression in the code. Symbolically, "$\not{\$}$D" definitions may take on one of the following forms (where X represents any letter of the alphabet or a blank):

1. $\not{\$}$DX = N, where "N" is a fixed or floating point literal or a parameter algebra expression. This is equivalent to the definition and loading of an array of length one discussed at the beginning of Chapter 3.

2. $\not{\$}$DX(P), where "P" is a parameter algebra expression. This entry sets aside a vector of length "P" and assigns to it the symbol "$\not{\$}$DX."

3. $\not{\$}$DX($P_1$,$P_2$,...,$P_n$), sets aside an n-dimensional block of length P_1*P_2*...*P_n and assigns to it the symbol "$\not{\$}$DX."

4. $\not{\$}$DX(P) = Q_1,...Q_n, or $\not{\$}$DX($P_1$,...,$P_n$) = Q_1,...Q_m, both define and load the vector or array specified. The "Q_i" may take on only the following forms: fixed point literal,

floating point literal, parameter algebra expression, set words to zero expression "(Z)P," or skip words expression "(S)P."

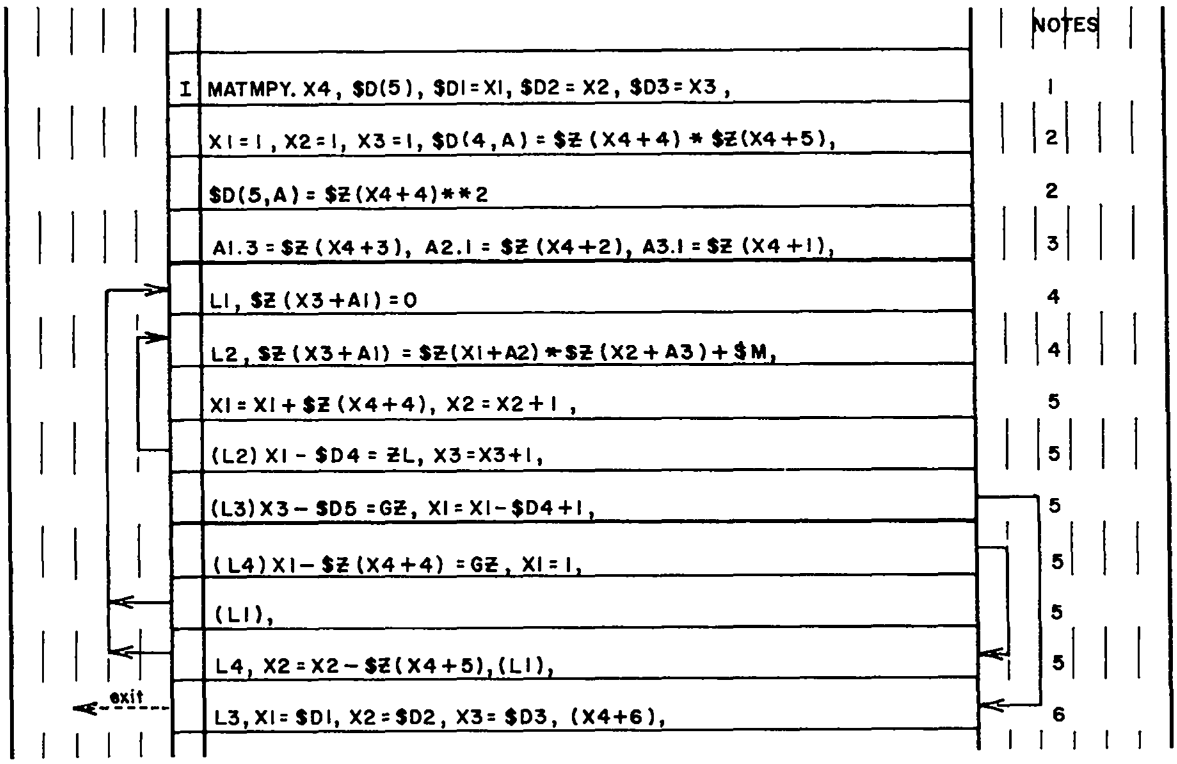
5. *$DX = SYMBØL, where "SYMBØL" represents some alphabetic symbol previously defined on "D" cards. This is the familiar "equivalence" feature, making the "$DX" block specified equivalent to the block named by "SYMBØL." Other features of data entry, such as interpolations, multiples, repeats, double-storing, etc., are not allowed in the entry of "$D" blocks. The equivalence feature, however, permits sophistications like this where necessary.

*IVY handles "$D" blocks as follows: when a formula set name (identified as such by its previous occurrence on a "D" card) is encountered during assembly, the "$D" control words are set to zero. If definitions of "$D" symbols then follow the formula set name, control words are constructed for the blocks defined, in the same manner as is done for symbols on "D" cards, space is allocated, and loading (if any) is performed. The control words thus constructed are now used in the assembly of the formula set to compute the proper addresses for instructions which refer to the "$D" blocks. Each formula within the given set thus has common access to the "$D" blocks defined for this set, but no formula outside the set can reference these "$D" blocks.

For example, let us consider a formula set which consists of a matrix multiply routine similar to that considered above, page 99. Let us call the routine MATMPY. Its function will be to multiply the two matrices "MA" and "MB" (the first of which has dimensions I,J), and place the result in "MC." We shall define the calling sequence as follows:

```
($P,MATMPY: MA($WA):MB($WA):MC($WA):I:J),
```

A routine which will perform the required matrix multiply is:



```
                                                                    NOTES
I  MATMPY.X4, $D(5), $D1=X1, $D2=X2, $D3=X3,                          1
   X1=1, X2=1, X3=1, $D(4,A)=$Z(X4+4)*$Z(X4+5),                       2
   $D(5,A)=$Z(X4+4)**2                                                2
   A1.3=$Z(X4+3), A2.1=$Z(X4+2), A3.1=$Z(X4+1),                       3
   L1, $Z(X3+A1)=0                                                    4
   L2, $Z(X3+A1)=$Z(X1+A2)*$Z(X2+A3)+$M,                              4
   X1=X1+$Z(X4+4), X2=X2+1 ,                                          5
   (L2) X1-$D4=ZL, X3=X3+1,                                           5
   (L3)X3-$D5=GZ, X1=X1-$D4+1,                                        5
   (L4)X1-$Z(X4+4)=GZ, X1=1,                                          5
   (L1),                                                              5
   L4, X2=X2-$Z(X4+5),(L1),                                           5
   L3,X1=$D1, X2=$D2, X3=$D3, (X4+6),                                 6
```

NOTES:

1. The block "$D" is defined as being five words long. X1, X2, and X3 are stored in the first three of these words. It is assumed that "MATMPY" is a formula set name.

2. I*J and I**2 are computed as in the previous example, and "A" is specified since the arithmetic is ambiguous. Floating point and Boolean numbers can be entered in a calling sequence.

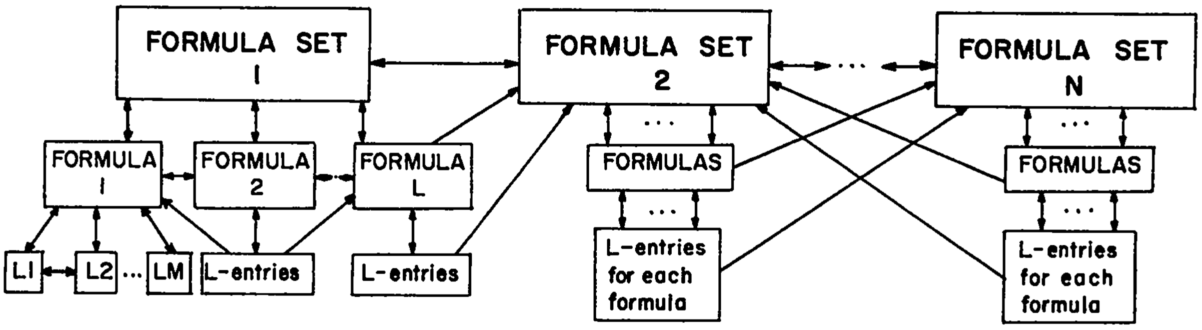
3. Examples of "store address" expressions. The base addresses of "MA," "MB," and "MC" are picked up from the calling sequence and stored properly.

4. Matrix "MC" is evaluated.

5. Index registers are altered and conditional branches are performed. "A" must be specified in all cases where dynamic modification is desired.

6. X1, X2, and X3 are restored and exit from the subroutine, to (X4+6), is performed.

Some final notes on the organization of instructions. The block diagram below should serve as an illustration of how a typical IVY code should be organized:



"Horizontal" branches are allowed between formula sets, formulas in the same set, and L-entries in the same formula. "Vertical" branches are allowed between a formula set and its formulas, and between formulas and their L-entries. "Diagonal" one-way branches are allowed between formulas and formula sets, and L-entries and formula sets or formulas in the same set. This diagram is a summary of all that has been said previously on the hierarchical organization of an IVY program.

*Before assembly, formula sets are written on a tape under control of an "A" card. Each formula set may be written separately, or a number of formula sets may be written together, as desired. The entire code may be assembled at one time, or the various portions may be assembled when they are needed. Even a code which contains too many instructions to fit a given machine may still be assembled and executed a package at a time. Practical examples on the usage of these features are included in Appendix 1.

<u>The execute instruction</u>. The execute instruction is the only entry allowed on "I" cards which has not yet been discussed. This instruction is an alternative way to cause IVY to transfer control to an assembled code without using the "X" card. The execute instruction may appear as the very last statement in a formula set, and when encountered by the assembly program, it causes IVY to give control to the formula set or to a formula in the set just completed. The format is as follows:

,$E.SYMBØL,

where "SYMBØL" represents the name of any previously converted formula set, or of a formula in the formula set which has just been assembled.

*<u>Summary</u>. The purpose of this chapter has been to introduce the programmer to the various IVY branching conventions and to the organization of an IVY program, and in connection with this, such topics as calling sequences, transfer tables, the manipulation of calling sequence words, the "$D" symbols for subroutine data, and the execute instruction. Our discussion of the IVY algebraic language is now complete except for the coding examples in Chapter 8. Table IV summarizes the types of IVY branches and entries discussed in this chapter and Chapter 4.

TABLE IV

IVY BRANCHING CONVENTIONS

1. Types of entries:

	ENTRY	MEANING	PAGE	TYPE OF BRANCH NEEDED
a.	,X_n(A,B),	Index loop entry	92	2-a.
b.	,L_n,	Local or L-entry	97	2-b, 2-c
c.	,L_n.Xn,	Local or L-entry subroutine	101	2-d
d.	,ꬾORM,	Formula name	102	2e,2-f
e.	,FORM.X_n,	Formula subroutine	102	2-g.
f.	,FS,	Formula set name	104	2-h, 2-i.
g.	,FS.X_n,	Formula set subroutine	104	2-j

2. Types of branches:

	BRANCH	TYPE	PAGE	TO ENTRY OF TYPE
a.	,(X_n),	End of index loop	92	1-a
b.	,(L_n),	Unconditional L-branch	97	1-b
c.	,(L_n,M) Algebra = C,	Conditional L-branch	98	1-b
d.	,(\$P,L_n),	Pathfinder L-branch	101	1-c
e.	,(FØRM),	Unconditional branch to formula	103	1-d
f.	,(FØRM,M), Algebra = C,	Conditional branch to formula	103	1-d

TABLE IV (Continued)

	BRANCH	TYPE	PAGE	TO ENTRY OF TYPE
g.	,(ØP,FØRM),	Pathfinder branch to formula	103	1-e
h.	,(FS),	Unconditional branch to formula set	104	1-f
i.	,(FS,M) Algebra = C,	Conditional branch to formula set	104	1-f
j.	,(ØP,FS),	Pathfinder branch to formula set	104	1-g
k.	,(X_n + 1),	Subroutine exit	101	return to 2-d, 2-g,2-j
	or			
	,(X_n + N + 1),	Subroutine exit if path-finder branch followed by N calling sequence words	106	

CHAPTER 6

IVY SUBROUTINES

Any of the IVY "$\not{\$}$" subroutines described in this chapter may be entered by means of a pathfinder branch from the coder's program. In some cases the routines require a variable length calling sequence; when this is so, it is to be understood that the calling sequence can occur either in the code, as follows:

```
,(₿P,₿RØUT: Calling sequence),
```

or on a "K" card, in which case the symbol representing the calling sequence block must appear in the instruction calling sequence, as follows:

```
,(₿P,₿RØUT: SYMBØL (₿WP))
```

The two techniques are equivalent provided that the two calling sequences are the same. Thus, when in the description of a particular "$\not{\$}$" routine, calling sequence entries are described, it is understood that these entries can appear either between colons after the pathfinder branch on the instruction card, or between colons on a "K" card which is addressed by a single calling sequence word entry after the pathfinder branch.

A. The loading program. The loading program "$\not{s}$LD" has no calling sequence and is entered by a simple pathfinder branch:

,($\not{s}$P,$\not{s}$LD),

"$\not{s}$LD" is the program which reads in cards from the on-line reader or the off-line input tape, recognizing and treating all the various types of IVY cards described in Chapter 2. If an end-of-file condition occurs in the reader or on tape, IVY halts after printing an appropriate comment. The use of the loading program is twofold:

1. To read in new data, on "E" cards, to read in and/or assemble new code, to read in an "Ø" card and halt temporarily, etc. In this case control is returned to the expression immediately following the pathfinder branch to "$\not{s}$LD" in the programmer's code when an "X" card with columns 2-72 blank is encountered, or to some formula set if an "X" card containing the name of this formula set is encountered. Control may also be returned if a "$\not{s}$E" entry is encountered in a formula set being converted, as described in Chapter 5, page 115.

2. To exit from a program when its execution has been completed. The last executable instruction in any IVY code should be a pathfinder branch to "$\not{s}$LD," in order to read in any program stacked behind the current one, or to halt if no such program exists.

B. The assembly program. The assembly program "$\not{s}$AP" can be entered internally to avoid reading an "A" card through "$\not{s}$LD." However, under programmer control "$\not{s}$AP" will read only, and will not write on tape. The calling sequence for "$\not{s}$AP" is:

($\not{s}$P,$\not{s}$AP: Calling sequence),

where the calling sequence may occur in the parentheses or on a "K" card, provided that in the latter case only the name of the "K" card and the modifier "$\not{\$}$WP" occur in the parentheses following the pathfinder branch to "$\not{\$}$AP." The calling sequence to "$\not{\$}$AP" may consist of any number of entries, each consisting of one calling sequence word, as follows:

$\not{\$}$RDN,F

where: "$\not{\$}$RD" means "read"; "N" is a hexadecimal expression for the tape number; and "F" is a parameter algebra expression for the file number on tape "N." If N = 0, the systems tape (equivalent to N = A) is used.

Any number of files on the tape may be read and converted using "$\not{\$}$AP"; however, IVY will give up control to any formula set which ends with an execute statement, as mentioned at the end of the previous chapter.

Table V shows the correspondence between the tape number "N" and channel and tape numbers on the IBM 7090. These correspondences hold for both "$\not{\$}$AP" and "$\not{\$}$TP." Also noted are the tapes which are reserved for the various types of off-line output. These tape numbers may not be used by "$\not{\$}$AP" or "$\not{\$}$TP" (see next section) unless precautions are taken to protect the output of previous jobs which may be on these tapes.

TABLE V

CORRESPONDENCE BETWEEN IVY TAPE NUMBERS AND 7090 TAPE NUMBERS

IVY TAPE NO.	7090	RESERVED FOR USE (IF ANY)
0	A2	Assembly, if no other specified
1	A1	IVY system (high density)
2	A5	None
3	B3	None
4	A4	None
5	B1	None
6	B2	None

TABLE V (Continued)

IVY TAPE NO.	7090	RESERVED FOR USE (IF ANY)
7	A6	None
8	B6	BCD Input, for reading decks off-line
9	A3	BCD Output (print & punch, high density)
A	A2	Assembly, if no other specified
B	B4	None
C	B5	4020 Output (plot & print, low density)

On the IBM 7030, tape numbers 1-F can be specified. However, no list of corresponding absolute channel and tape numbers can be given since these are assigned by IVY on the basis of available tapes and the requirements of other programs. In general, the programmer or operator will be notified of these assignments via the console typewriter in advance of the time they are used by the program, so that tapes may be properly mounted and the dials set in plenty of time.

C. Tape manipulation program. "$TP" is a program allowing complete flexibility in manipulating binary (i.e., odd parity) tapes. Its features include reading, writing, spacing forwards or backwards, positioning, setting density, rewinding, unloading, writing end-of-file, and testing the current tape, all under control of various calling sequence words. The calling sequence is designed in such a manner that each calling sequence word, as a rule, represents one simple mnemonic instruction to a particular tape unit.

*For the benefit of those who are not familiar with IBM tape operation, a short summary is included here. The shortest block of information written on a tape is a record, consisting of one or more words of data, and separated from other records by a gap in the tape called the end-of-record gap. Records, in turn, may be gathered together into files; files are separated from one another by a record consisting of a special character followed by a long gap on the tape, called the end-of-file gap or simply the end-of-file. In addition, IVY marks the end of the tape with a special record called the end-of-tape record, consisting of one word containing special information and an end-of-file.

Each record read or written by "$TP" must be preceded by an identification word, called the "ID." Each record in a file must have an ID different from the other records in the same file, in order to facilitate searching procedures. When "$TP" is searching for a record with a particular ID, it searches only the file in which the tape is positioned when this record is requested. In no case will "$TP" ever read beyond the end-of-tape record, or write beyond the end-of-tape reflective strip.

The calling sequence to "$TP" may consist of any number of entries, each of which may be any one of the following:

1. $HDX, where "X" is a tape number in hexadecimal, $1 \leq X \leq C$ on the 7090, $1 \leq X \leq F$ on the 7030. This causes tape "X" to be set to high density (556 bits per inch).

2. $LDX. Tape "X" is set to low density (200 bits per inch).

3. $RWX. Tape "X" is rewound to the load point. (Identical to performing manually the operations of pressing the "reset," "load rewind," and "start" buttons on the tape unit.)

4. $ULX. Tape "X" is rewound to the load point, the upper head assembly is raised, and the tape is removed from the vacuum columns. (Identical to performing manually the operations of pressing the "reset," "load rewind," and "unload" buttons on the tape unit.)

5. $EFX. Write an end-of-file mark on tape "X."

6. $ETX. Write the end-of-tape record on tape "X," and backspace the tape to the beginning of this record.

7. $BBX,P, where "P" represents a parameter algebra expression.

This instruction causes tape "X" to backspace through "P" blocks or records, where an end-of-file is counted as a record. Error indication is given if "P" is large enough to cause the tape to attempt to backspace through the load point. If P = 0 this instruction is ignored.

8. $BFX,P. Tape "X" is backspaced over "P" files, where the count "P" includes the current file. The tape is then positioned to read the first record of the file located. Error indication is given if "P" is large enough to cause the tape to attempt to backspace through the load point. If P = 0,this instruction is ignored. If P = 1, the tape is set to read the first record of the current file.

9. $FBX,P. Tape "X" is spaced forward over "P" blocks or records. Error indication is given if "P" is large enough to cause the tape to attempt to space forward through an end-of-file, or if the tape is positioned at the end-of-tape record. If P = 0, this instruction is ignored.

10. $FFX,P. Tape "X" is spaced forward over "P" files, where the count "P" includes the file in which the tape is currently positioned. An attempt to space beyond the end-of-tape record will cause an error indication. If P = 0, this instruction is ignored.

11. $RDX,AD($WA)+P, where "AD" is any programmer symbol and "P" is a parameter algebra expression. This entry may occur <u>only</u> at the end of a calling sequence. The ID of the record at which tape "X" is positioned is compared to the contents of the location specified by "AD($WA)+P," and if the two are equal, $CS1 is set to 1. If they are not equal, $CS1 is set to 0. The tape remains positioned to read this record.

<u>NOTE</u>: The next two entries each consist of a <u>pair</u> of calling sequence words.

12. $RDX,AD($WA)+P: AE($WP). "$TP" attempts to find the record in the current file on tape "X" with ID equal to the contents of "AD($WA)+P," and if successful, reads the record into block "AE." The current file is scanned twice in an attempt to find the record with the specified ID, so to save time the programmer should attempt to position the tape at the proper record before giving the "read" command. If the proper record cannot be found, error indication is given and "$TP" relinquishes control to IVY, which selects the card reader or input tape in an attempt to find the

next job. "$TP" does not return control to the programmer's calling sequence, since the lost data may have been essential to the program. The block "AE" must have a non-zero count and base address, and the count of the record read from tape may not exceed the count of "AE." Otherwise, error indication is given and IVY takes control. (Using option 11 plus the proper spacing instructions the programmer can easily locate the proper record before reading it.)

13. $WRX, AD($WA)+P: AE($WP). A record is written on tape "X," starting at the point where the tape is positioned, and containing the ID specified by "AD($WA)+P" followed by the contents of block "AE." If the physical end-of-tape is sensed at any point in the writing of this record, the tape is backspaced to the start of the record, an end-of-tape record is written, and the machine halts after printing a comment to this effect. When a new tape has been mounted, and "start" (7090) or "console signal" (7030) has been pressed, "$TP" will write the record on the new tape. The programmer should note that when a new record is written on a tape containing other information, any of the old information beyond the new record will become unreadable. Thus, the new record should be written <u>after</u> any information that is still needed.

14. $PØ. This calling sequence word causes parallel operation to take place during all the tape operations specified in subsequent calling sequence words until "$SØ" is encountered (see below). By parallel operation it is meant that the tape input-output will proceed in parallel with computation. This entry should be used only in those cases where subsequent computations do not depend on the results of the operation, e.g., when subsequent computations do not address a block which is being read in or written, etc. Care should be exercised in the use of this entry, since it makes detection of RTT and end-of-tape somewhat more difficult.

15. $SØ. This calling sequence word specifies serial operation, i.e., each input-output instruction is completed before the next is initiated, and all operations are completed before "$TP" returns control to the programmer's code. If neither "$SØ" or "$PØ" is specified in the calling sequence, serial operation is always performed.

*<u>Detection and treatment of errors by "$TP."</u> Two types of errors are detected by "$TP": Errors arising from the tape unit itself (redundancy errors), and programming errors. These are treated as follows:

1. Redundancy on writing: A backspace is performed and an attempt is made to rewrite the record. If no error is detected on the second writing, "$TP" proceeds without error indication. If a second redundancy error is detected, error indication is given, a gap is erased on tape, and a third attempt is made to write the record. If this attempt is still unsuccessful, the machine stops after printing a comment that the tape is defective and should be replaced. After the tape has been replaced, press "start" or "console signal" and the program will proceed.

   Redundancy on reading: At most ten attempts are made to read the faulty record. If all are unsuccessful, an error indication is given and control is surrendered to IVY, which selects the card reader or input tape in an attempt to find the next job.

2. Programming errors include such things as attempting to space the tape too far forwards or backwards, asking to read a record which is not in the current file, incorrect calling sequence words, etc. In general, spacing errors will cause an error indication, but "$TP" will proceed to the next calling sequence word. Errors affecting actual reading and writing operations will cause error indication, and "$TP" will surrender control to IVY, since it is assumed that the tape input or output operation is vital to the code, and that the code cannot run without it. Some errors, e.g., end-of-tape detection, cause a halt; when the condition has been corrected, pressing "start" or "console signal" will cause "$TP" to proceed. "$TP" indications are included in the list of IVY error indications.

Some examples of "$TP" calling sequences: Included also is the action "$TP" takes under various conditions.

EXAMPLE	NOTE
($P, $TP: $S∅),...	1
($P,$TP: $HD3: $WR3, GE($WA)+1: FXNTZ($WP): $BB2,3: $RD2, TH($WA)+2)	2
($P, $TP: $P∅: $RW5: $WR5,FN($WA)+1: AD($WP): $WR5, FN($WA)+2: AE($WP): $WR5, FN($WA)+3: AF($WP):$EF5)	3
($P, $TP: $FFB,3: $FBB,2: $RDB, RSN($WA)+2: AXX($WP))	4

NOTES:

1. This calling sequence entry simply causes the machine to wait until all input-output channels containing tape units have completed their current operation. An entry of this type can be used before operations affecting a block being read or written in the parallel mode are performed, so that a block still engaged in input-output transmission will not be altered before the transmission is completed.

2. In this entry, block "FXNTZ" is written in high density on tape 3, after which tape 2 is backspaced two records and the ID of the new record is checked. It is assumed that the density of tape 2 has already been set. Note that the density of a tape unit can be set internally using calling sequence words, or externally by pressing a button on the tape unit. In any event, once a tape is set to a certain density, this setting should not be changed. A given tape unit should always be read, written, backspaced, etc., in the same density.

3. The blocks "AD," "AE," and "AF" are written on tape 5, then an end-of-file is written. This is all done in the parallel mode, so that the operations may not be complete when "ØTP" returns control. An entry of the type discussed in note 1 should be used before any attempt is made to alter the blocks "AD," "AE," and "AF".

4. Tape "B" is spaced forwards 3 files plus 2 records, and block "AXX" is then read. It is always a time-saving procedure to position a tape properly before a record is read.

D. The switch test program. "ØSW," has no calling sequence and is entered by a simple pathfinder branch

(ØP,ØSW),

The purpose of "ØSW" is to test the six sense switches and read the keys on the 7090 console, and to read the settings of the various buttons, keys, and switches on the 7030 console. The information gleaned from the reading of these console devices is set up in the "ØCS" block in a format wnicn can be easily tested internally. The first six binary switches on the 7030 console are treated the same as the sense switches on the 7090 console to provide analogous input. "$\not{C}$CS" is set up as follows on the two machines:

TABLE VI

"$\not{C}$CS" BLOCK SETTINGS BY "$\not{C}$SW"

"SCS" WORD	CONTENTS, 7090	CONTENTS, 7030	TYPE OF WORD
$\not{C}$CS1	sense switch 1: 0 if off, 1 if on	binary switch 0	fixed
$\not{C}$CS2	sense switch 2	binary switch 1	fixed
$\not{C}$CS3	sense switch 3	binary switch 2	fixed
$\not{C}$CS4	sense switch 4	binary switch 3	fixed
$\not{C}$CS5	sense switch 5	binary switch 4	fixed
$\not{C}$CS6	sense switch 6	binary switch 5	fixed
$\not{C}$CS7	switches 1-3 in binary	switches 0-2	fixed
$\not{C}$CS8	switches 4-6 in binary	switches 3-5	fixed
$\not{C}$CS9	switches 1-6 in binary	switches 0-5	fixed
$\not{C}$CS10	console keys S,1-35	binary keys,0-63	Boolean
$\not{C}$CS11	zero	numerical switches	Boolean
$\not{C}$CS12	zero	first half word: binary switches. second half word: digital potentio-meters.	Boolean

Thus we note that ∅CS1-∅CS9 are analogous on both machines if we interpret the first six binary switches on the 7030 as equivalent to the 7090 switches. Similarly, the last 36 binary keys on the 7030 are equivalent to the 36 console keys on the 7090. The same sequence of instructions can be used on either machine to handle these 36 settings. If the other settings on the 7030 are used, one can retain compatibility by coding an alternative routine under the control of "∅L5," the machine number indicator. Or a program can be coded to test the 7030 settings without worrying about incompatibility with the 7090, if desired. Such a code, of course, will not function properly on the 7090 if compatibility is required.

E. The test trigger routine, "∅TT," is used to test the status of various internal machine indicators (ac overflow, divide check, etc., on the 7090, and various maskable indicators on the 7030). After the test all indicators are turned off. Certain words of "∅CS" are set according to the status of the indicators, and if desired, a diagnostic comment is printed on-line. Also, optionally, the programmer can cause "∅TT" to give up control to IVY if any of the indicators are on. IVY contains various internal programs to handle cases of floating point overflow or underflow, so in general the trigger settings will reflect only the results of fixed point operations. The trigger settings, and the resulting settings of "∅CS," are as follows:

TABLE VII

"ϕCS" SETTINGS BY "ϕTT"

"ϕCS" WORD	7090 TRIGGER	7030 TRIGGER	TYPE OF WORD
ϕCS1	AC overflow: 1 if on, 0 if off	LC,PF,LS: 1 if any is on	fixed
ϕCS2	MQ overflow: 1 if on, 0 if off	PSH: 1 if on, 0 if off	fixed
ϕCS3	Divide check: 1 if on, 0 if off	ZD: 1 if on, 0 if off	fixed
ϕCS4	Negative square root: 1 if negative, 0 if not	IR: 1 if on, 0 if off	fixed
ϕCS5	Zero always	indicator word: contents on entry to "ϕTT"	Boolean

Note that on the 7030 the contents of the entire indicator word are placed in ϕCS5 so that individual indicator bits may be tested. The indicator register is always set to zero on exit from "ϕTT."

The calling sequence to "ϕTT" is as follows:

$$(\phi P, \phi TT: \phi^{P}_{N}, \ AD(\phi WP): \ \phi^{R}_{I}), \ldots$$

where "ϕN" means "no print," "ϕP" means "print a diagnostic comment headed by the symbol 'AD' and containing a list of the indicators which were on"; "ϕR" means "return control to the problem program," and "ϕI" means "return control to IVY if any of the tested triggers were on." Note that "ϕTT" always has a fixed-length calling sequence two words long.

Only the format of these calling sequence words is variable. This calling sequence, being fixed length, cannot appear on a "K" card.

F. The octal dump program."$ØD" is used to obtain a dump in octal of any or all of the data blocks or longhand formula sets used in a program. A decimal dump of data, of course, can be obtained using the print program "$PR" described in the next section. The octal dump appears off-line on tape 9 unless console key 35 (7090) or binary key 63 (7030) is down, in which case the dump will be printed on-line. The calling sequence to "$ØD" may consist of one word (if a dump of all data regions and longhand formula sets is desired) or of any number of words if a dump of only certain blocks is desired. The one-word entry for dumping all blocks is:

($P,$ØD:$DA),...

where "$DA" is mnemonic for "dump all." If only certain blocks are to be dumped, the format of each calling sequence word is

...:SYMBØL($WP):...

where "SYMBØL" is the name of any data block or longhand formula set.

The dump consists of the following information:

1. The symbol for the block whose dump appears below;

2. A number of lines consisting of the contents of the block, in octal, each line being constructed as follows:

   a. A five-digit octal number giving the location of the first word on the line on the 7090, a six-digit number on the 7030;

   b. A string of octal numbers:

(1) eight numbers of twelve digits each are printed on the 7090;

(2) eight numbers each consisting of eight octal digits plus two hexadecimal digits in the standard format (i.e., four full words) are printed on the 7030.

G. The print program. "$PR" is used to print numbers in decimal in a wide variety of formats. This printing can be done, in a limited amount, on-line; or off-line on tape 9 for later printing on a peripheral device; or off-line on tape C for producing a listing on microfilm via the SC-4020 microfilm device. The format statement controls not only the format of individual numbers within each block printed, but also the arrangement of vectors and blocks on a page, the printing of column and row headings, remarks, etc. The "$PR" calling sequence has a large number of calling sequence words which will be explained in turn.

1. Spacing. The simplest calling sequence word entries control the spacing of the page. Normally single spacing takes place between lines and double spacing between blocks. If other spacing is desired between blocks, one or more of the following calling sequence word entries are used:

CSW	OPERATION
$1P	Restore page (on-line)
$2P	Half-page skip (on-line)
$DP	Double-space printer (on-line)
$1T	Restore page (tape 9)
$2T	Half-page skip (tape 9)
$DT	Double-space page (tape 9)
$1M	"Restore page" (advance film — tape C)

CSW	OPERATION
$2M	"Half-page skip" (microfilm — tape C)
$DM	"Double space" (microfilm — tape C)

2. Printing of remarks. Remarks may be printed, without the benefit of a format statement, by use of one of the following calling sequence word entries:

$P,REM($WP)	print remark on-line;
$T,REM($WP)	write remark on tape 9;
$M,REM($WP)	write remark on tape C for microfilm;

where "REM" represents a symbol for any remark. Remarks can also be printed, if desired, under the control of a format statement as described below.

3. Format statements. Format statements are remarks constructed in a particular manner to control the printing of arrays of numbers and "comment" remarks, as well as giving various other information about the format of the printout. The calling sequence entry for a format statement is as follows:

$F,FØRMAT($WP)

where "FØRMAT" is the name of the desired "format" remark statement. The format named then holds for all the following information until it is exhausted, a term which will be explained below.

The first field, and only the first field, of a remark statement must contain one or more control characters followed by a comma;

the general appearance of this part of a format statement is as follows:

$$R|F\emptyset RMAT = C_1C_2C_3...C_N,...$$

where "FØRMAT" is the name of the statement and the "C_i" are control characters. Each "C_i" may be any one of the following:

P	print on-line.
T	print off-line (tape 9).
M	print on microfilm (tape C).
C	print column indices on vectors or arrays.
R	print row indices on vectors or arrays.
L	print vectors and arrays in line format.
F	print fixed point numbers as integers regardless of format statement.
Ø	print fixed point tags of double-stored numbers in octal.
0 or blank	ignored.

The functions of the characters "C," "R," "L," "F," and "Ø" will be discussed somewhat later in this section. Of the characters "P," "T," and "M," only <u>one</u> may occur to specify the mode of printing desired. If more than one occurs, the last one in sequence will take precedence. At least one of these characters must occur; if <u>none</u> occurs, "T" will be assumed. The characters "T" and "M" can be overruled temporarily by console key 35 (7090) or binary key 63 (7030). If this is down, the next block encountered will be printed on-line regardless of the format statement. If key 35 is up, "T" or "M" will again take control. We thus note that this particular key is <u>always</u>

reserved to specify on-line printing; in general its use should be restricted to emergencies to save machine time.

To print a remark under control of a format statement, the following two calling sequence words must be used:

```
$F,FØRMAT($WP): REMARK($WP)
```

The remark will be printed on-line, off-line, or on the microfilm tape according to the setting of the print key and to which of the characters "P," "T," or "M" occurs in the format statement. For example, if "FØRMAT" and "REMARK" are defined as follows:

```
R|FØRMAT = P,$$$
R|REMARK = THIS IS A$$TWØ LINE REMARK.$$$
```

the above two calling sequence word entries will cause

```
THIS IS A
TWØ LINE REMARK.
```

to be printed on-line. ("$PR" automatically detects the "$$" and causes the printer to space at that point).

4. Printing numbers and arrays of numbers.

a. The "C," "R," amd "L" controls. As remarked above, "C" and "R" occurring among the control characters of a format statement cause column and row indices, respectively, to be printed with arrays of numbers. The first column index, if any, will always be replaced by the name of the block which is about to be printed; the rest of the column indices and the row indices ascend in sequence. These indices appear as follows:

```
               row
                ↓
      column →        NAME   2   3   4   ...   N
                1
                2
                3
                .
                .
                .
                M
```

The column indices are always properly centered over the appropriate column.

If "L" is not given, the so-called normal form of printing occurs: namely, one-dimensional arrays or <u>vectors</u> are printed in columns, and multi-dimensional arrays are printed in matrices such that the first index varies along columns. The following diagram illustrates the normal form of a vector and a two-dimensional array:

$$\begin{array}{c@{\qquad}ccccc}
\underline{\text{VECTOR}} & & \multicolumn{4}{c}{\underline{\text{TWO-DIMENSIONAL ARRAY}}} \\
A_1 & B_{1,1} & B_{1,2} & B_{1,3} & \cdots & B_{1,k} \\
A_2 & B_{2,1} & B_{2,2} & B_{2,3} & \cdots & B_{2,k} \\
A_3 & B_{3,1} & B_{3,2} & B_{3,3} & \cdots & B_{3,k} \\
\vdots & \vdots & & & & \vdots \\
A_n & B_{j,1} & B_{j,2} & B_{j,3} & \cdots & B_{j,k}
\end{array}$$

N-dimensional arrays (where $N \geq 3$) are printed as two-dimensional matrices. In each matrix the last N-2 dimensions are constant, and the first index varies along columns as above. The matrices are printed in such an order that the first of the last N-2 dimensions varies most rapidly from matrix to matrix, the second dimension less rapidly, ..., the Nth dimension least rapidly. Double spacing always occurs between the matrices of an array. Thus, for example, if array "C" has dimensions (I, J, K, L,...,P) and $M_{k,\ell,\dots,p}$ represents one of its two-dimensional matrices, these matrices are printed in the following order:

$M_{1,1,\dots,1}$

double space

$M_{2,1,\dots,1}$

double space

$\vdots$

$M_{k,1,\dots,1}$

double space

$M_{1,2,\dots,1}$

double space

$M_{2,2\dots,1}$

double space

$\vdots$

$M_{k,\ell,\dots,p}$

double space

$M_{k+1,\ell,\ldots,p}$

double space

$\vdots$

$M_{K-1,L,\ldots,P}$

double space

$M_{K,L,\ldots,P}$

If the rows of any two-dimensional matrix are so long as to exceed the size of the page (119 columns), the matrix will then be printed as follows:

$$\begin{array}{lllll}
B_{1,1} & B_{1,2} & B_{1,3} & \cdots & B_{1,i} \\
B_{2,1} & B_{2,2} & B_{2,3} & \cdots & B_{2,i} \\
\vdots & & & & \\
B_{j,1} & B_{j,2} & B_{j,3} & \cdots & B_{j,i}
\end{array}$$

double space

$$\begin{array}{lll}
B_{1,i+1} & \cdots & B_{1,k} \\
B_{2,i+1} & \cdots & B_{2,k} \\
\vdots & & \\
B_{j,i+1} & \cdots & B_{j,k}
\end{array}$$

Since the overflowing portion of the matrix is printed separately, it can be read easily as an extension of the first portion. Its *row* indices (if any) are reset to start at 1 and its column indices (if any) continue from the highest previous value. Thus, if desired, the later portion of the listing can be cut out and attached to the right of the earlier portion to give a complete picture of the array.

If an "L" is given in the format statement, the so-called *line form* of printing occurs; that is, vectors are printed *across* the line, and multi-dimensional arrays are printed so that the first index varies in *rows*, as follows:

$$\begin{array}{llllll}
\text{VECTORS} & A_1 & A_2 & A_3 & \cdots & A_N \\
\text{ARRAYS} & B_{1,1} & B_{2,1} & B_{3,1} & \cdots & B_{k,1} \\
 & B_{1,2} & B_{2,2} & B_{3,2} & \cdots & B_{k,2} \\
 & B_{1,3} & B_{2,3} & B_{3,3} & \cdots & B_{k,3} \\
 & \vdots & & & & \\
 & B_{1,j} & B_{2,j} & B_{3,j} & \cdots & B_{k,j}
\end{array}$$

The comments given above about the order of printing of the matrices of an N-dimensional array and the convention in case the rows of a matrix overflow the page also apply here.

b. *Format control of numbers*. The control characters mentioned above govern only the outward appearance of a listing: the spacing between blocks, the appearance of row and column indices, and the order in which the elements of a vector or array appear across the page. The remainder of the format statement specifies the exact appearance of each individual number in the listing. After the control characters have appeared, any number of fields consisting of five decimal numbers separated by periods may occur, each specifying the format for one or more blocks of data. This appears as follows:

$$\mathrm{R|FØRMAT} = C_1C_2\ldots C_N,\ \mathrm{N.S.I.F.E.,\ etc.}$$

"N," "S," "I," "F," and "E" are decimal numbers having the following meanings:

N:	The number of blocks of data for which this portion of format is to be used.
S:	The number of blank spaces which precede each number printed under control of this format statement.
I:	The number of integer digits (to the left of the decimal point) to be printed in the numbers under control of this portion of the format.
F:	The number of fractional digits (to the right of the decimal point) to be printed.
E:	The number of exponent digits to be printed.

For example, if "F1" is a format as follows:

```
R|F1 = P, 1.1.1.7.2, $$$
```

and if "B" is a vector of data, the <u>two</u> calling sequence words

```
$F, F1($WP): B($WP)
```

will cause the elements of the vector "B" to be printed on-line in a column as follows:

```
_ -x.xxxxxxx±xx
```

where "x" represents a decimal digit, "_" represents a blank space, "-" represents a minus if the sign is negative or blank if positive.

A right-to-left dropout feature operates in the "N.S.I.F.E." fields. That is, if ".E" is omitted, the number will be properly adjusted and printed without an exponent; if ".F.E" is omitted, the number will be printed as an integer; and if ".I.F.E" is omitted, "S" spaces will be inserted in the listing "N" times. Any combination of the numbers "S," "I," "F," or "E" may be zero to work up variations on this theme.

In general, the "N.S.I.F.E" field holds for all numbers in the block printed under its control regardless of whether they are fixed or floating point. However, if "F" appears among the control characters discussed in the previous section, all fixed point numbers in the block will be printed as integers, regardless of whether ".F" and ".E" are zero or not. The number of digits printed will equal I+F+E+2, subject to zero print control (i.e., lead zeroes are suppressed).

The general calling sequence entry then for printing M vectors under the control of a single format statement is:

$\not{\$}$F,FØRMAT ($\not{\$}$WP): VECTØR$_1$($\not{\$}$WP): VECTØR$_2$($\not{\$}$WP):...:VECTØR$_M$($\not{\$}$WP)

The format statement must be sufficient to print all M vectors; i.e., the sum of the N's in all the "N.S.I.F.E" fields in which at least one of I, F, and E is non-zero must be at least M. For example, the following format will serve to print 7 vectors (or less):

```
R|F2 = PCR, 3.1.1.7.2, 3.2, 2.1.1.5, 2.2.2.6.1, $$$
```

To print a string of parameters starting with a given parameter under control of some format statement, the following calling sequence entry is used:

```
$F, FØRMAT($WP): $PN, PARAM($WA)+P
```

where N $\leq$ 99 is the number of parameters desired and "P" represents parameter algebra the result of which must be at least 1. Then the N consecutive parameters starting with the one in location PARAM($WA)+P are printed in the same manner as a vector.

If the programmer wishes to print the parameters in several blocks using this single entry, the name "PARAM" must specify the last of these blocks loaded on "D" cards. The parameter blocks are then printed in reverse order of their loading, from last to first; the numbers in each block, however, are printed in sequence from first to last.

For example, consider the following format statements and calling sequence to "$PR":

```
R|F1 = CRT, 3.2.1.5.2, 2.0.3.2, $$$
R|F2 = CRPL,  1.1.2.3.1, $$$
```

and

```
I|(ØP, ØPR:ØF,F1(ØWP):MXT(ØWP):APX(ØWP):FNTZ(ØWP):
 |TNPRL(ØWP):VLTN(ØWP):ØF,F2(ØWP):ØP5,PC(ØWA) + 1),...
```

The vectors "MXT," "APX," and "FNTZ" will be printed in parallel columns in the format

$\underline{\ \ }$ -x.xxxxx±xx,

the two vectors "TNPRL" and "VLTN" will be printed in columns parallel to these in the format

$\underline{\ }$ -xxx.xx,

and five parameters starting with "PC" will be printed across a subsequent line in the format

$\underline{\ }$-xx.xxx±x.

Printing multi-dimensional arrays presents a somewhat more difficult problem since the first one or two dimensions must be made known to the print program in order to break the block up into its correct two-dimensional sub-matrices. To print a two-dimensional array, or matrix, assuming a previous format statement, the following is necessary:

```
Ø2, MATRIX(ØWP):P
```

where "MATRIX" is the symbol for the block and "P" is a parameter algebra expression for the first dimension. To print an array of three or more dimensions, the following entry is used:

ØA, ARRAY(ØWP): P_1:P_2

where "ARRAY" is the symbol for the block and "P_1" and "P_2" are parameter algebra expressions for the first two dimensions. If these entires are not used for multi-dimensional blocks, the blocks will be printed as vectors.

Arrays or vectors (but not parameter blocks) may have been defined as double-stored numbers. If this is so, and one wishes to print both the "Q" and "T" portion of these numbers, an extra calling sequence word giving the tag length must be placed immediately after the one naming the block, as follows:

vector:	VECTØR(\$WP):\$D,P
matrix:	\$2,MATRIX(\$WP):\$D,P_1:P_2
multi-dimensional array:	\$A,ARRAY(\$WP):\$D,P_1:P_2:P_3

where "\$D" is mnemonic for "double-store," and the parameter algebra following it is the tag length in bits. The other parameter algebra expressions are for the dimensions as described above. When double-stored numbers are printed in this fashion, the tag, in decimal (or octal if "Ø" was one of the control characters in the format statement described on page 133), preceded by the letter "T," occupies the low order digits of the fraction. Thus, for example, if a double-stored block contains a five-bit tag and is printed according to the format "N.1.1.7.2," each number of the block will appear as follows:

$\underline{\ }$ -x.xxxxTDD±xx

where "D" represents an octal or decimal tag digit. The coder should bear this in mind and adjust the size of ".F" accordingly.

Under control of one of two other calling sequence words, one can print only the "Q" or only the "T" portion of a double-stored number, if desired. The calling sequence entries to do this are:

	print "T" only:
vector:	VECTØR($WP):$T,P
matrix	$2,MATRIX($WP):$T,$P_1$:$P_2$
multi-dimensional array:	$A,ARRAY($WP):$T,$P_1$:$P_2$:$P_3$

(Regardless of the format statement, the tag is always printed as a decimal or octal integer preceded by the letter "T.")

	print "Q" only:
vector:	VECTØR($WP):$Q,P
matrix	$2,MATRIX($WP):$Q,$P_1$:$P_2$
multi-dimensional array:	$A,MATRIX($WP):$Q,$P_1$:$P_2$:$P_3$

5. Printing immediate remarks. We have already discussed how to print ordinary remarks. Immediate remarks, however, differ from ordinary remarks in that they occur in the format statement itself, and may precede any "N.S.I.F.E" field enclosed in parentheses, thus:

,(REMARK)N.S.I.F.E,

The characters in an immediate remark may be any legal hollerith characters, except that parentheses occurring in the immediate remark must be closed, i.e., must occur in the order

(...).

Immediate remarks should be reasonably brief (in no case, over 119 characters long) and are used usually for short headings, etc., and permit a considerable sophistication in the printout. A pair of examples will serve to illustrate possible uses:

```
R|F1=P,(PARAM.=) 1.0.1.7.2, $$$
```

and

```
I|($P,$PR:$F,F1($WP):$P1,AD($WA)+1),...
```

will cause parameter "AD1" to be printed as follows:

```
PARAM.=-x.xxxxxxx±xx
```

The following format:

```
R|FF=CRT,1.54, (TEMPERATURE) 1.54,1.1.1.5.2, $$$
```

and the calling sequence

```
I|($P,$PR:$F,FF($WP):$2,AA($WP):GE+3),...
```

will cause the heading

TEMPERATURE

to appear centered in the page above the printout of matrix "AA."

*6. Error detection. "$PR" detects a variety of errors and prints out a number of diagnositics on-line. Control is always returned to the problem program, however, since the validity of the output print has no effect on the workings of the program. Some of the more serious errors are:

a. Attempting to print blocks of numbers with no format statement;

b. Attempting to print blocks with an inadequate format statement, i.e., one which does not have enough non-zero ".I.F.E" fields to cover every block specified;

c. Attempting to print an undefined block, i.e., one for which the symbol is not in the symbol table, or which has no address defined;

d. Attempting to print a block of code.

Other less serious errors, for which no indication is given, are:

a. Attempting to print both the "Q" and "T" portions of a double-stored number, when ".F" is not large enough to include the entire tag and the letter "T." In this case, printing of the tag is suppressed.

b. Lost significance, e.g., in a case where ".E" is zero or missing, an integer part is too large for the ".I" specified. In this case the integer is printed modulo 10^I. Or specifying too few digits for the exponent, in which case the exponent is printed modulo 10^E.

H. The Punch Program, $̸PH, is quite similar in usage and conventions to the print program. The calling sequence and format statements are simply a subset of those for the print program. "$̸PH" produces "R" cards and "E" cards which may be used as input to a future IVY program. The following differences exist between "$̸PH" and "$̸PR":

1. No microfilm option is allowed in "$̸PH." If the "M" control character appears in the calling sequence or in a format statement, it is replaced with "T." (Note that cards written off-line for later punching are also placed on tape 9 with a distinguishing character detected by the IBM 1401 in the output process.)

2. If any of the spacing options (e.g.,"$̸1P," "$̸2P,"etc.) occur in the calling sequence, a blank card is produced. Blank cards are ignored by "$̸LD."

3. The control characters "C," "R" and "L" of the format statement are inoperative. Numbers of a block are simply punched in sequence on "E" cards separated by commas after the symbol for the block and an equal sign.

4. Whether or not the "F"control character is present in a format statement, fixed point numbers are punched as integers so that the "E" cards are punched properly for input usage.

5. Remarks are punched on "R" cards (and continuation cards, if necessary) complete with name, equal sign, and all necessary "$$" and "$$$" characters.

6. Parameters are not punched using the "$PN" convention, but must be punched as vectors.

7. "$PH" ignores immediate remarks in its controlling format statements.

8. Double stored numbers are punched in the proper input format, i.e.,

$$AD(Q.P) = N_1,N_2,\cdots,AD(T.P) = M_1,M_2,\cdots$$

and so on. The "Ø" flag of the format statement is still operative and will cause the tags to be punched in octal, prefixed of course by the "(B)" entry described in Chapter 3.

Bearing these differences in mind, the programmer can easily punch data and remarks using essentially the same techniques used for printing. In fact, the same format statements can be used for both punching and printing without error indication from "$PH."

I. The microfilm plot program. "$MP" is used to produce output on tape C which, when used as input to the SC-4020 peripheral microfilm device, will cause graphs to be produced on microfilm. We have already seen, in the section describing "$PR," how ordinary output listings can be produced by this device. Plotting is a somewhat more complicated business; it is not, however, the purpose of this writeup to describe the characteristics of the 4020 in detail. Instead, the curious reader is invited to read either the 4020 manual itself or any of a variety of writeups on the 4020 which are available.

"$\not{S}$MP" permits a wide variety of operations: advance film, select grid, label grid, write a label horizontally or vertically, plot points with an option of connecting points with a vector, and generate an arbitrary set of axes. This is all done under control of various calling sequence words as described below.

1. To advance film, the following calling sequence word must be used:

   $\not{S}$AFN,P

   where "N" is either 0 or 1: 0 if no hard copy is desired, 1 if hard copy is desired. This calling sequence word causes the microfilm to be advanced one frame; if N = 1, a series of vertical lines are drawn signifying that hard copy is desired; the film is then advanced "P" more frames where "P" is a parameter algebra expression. For example, the entry

   $\not{S}$AFO

   will simply cause the film to be advanced one frame. The film must be advanced whenever a graph is complete.

2. The calling sequence words specifying a "select grid" cause a grid of lines to be drawn horizontally and vertically for the graph coming up. The normal film frame has 1023 plotting positions in each direction, and the grid is drawn in the upper right corner of the frame, 900X900, to allow room at the left and below for labeling. A number of types of grids are available under calling sequence control. The following <u>three</u> calling sequence words are used to select a grid:

   $\not{S}$SG: P_1:P_2

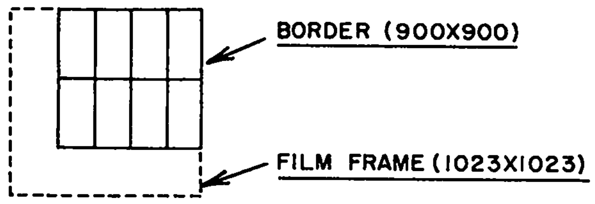
   where "P_1" and "P_2" are parameter algebra expressions the value of which specify the horizontal option and vertical option respectively, as follows:

VALUE OF P_1	GRID OPTION
1	1 interval (border only)
2	2 intervals (linear)
3	3 intervals (linear)
4	4 intervals (linear)
5	5 intervals (linear)
6	10 intervals (linear)
7	15 intervals (linear)
8	20 intervals (linear)
9	25 intervals (linear)
10	50 intervals (linear)
11	1 cycle logarithmic
12	2 cycle logarithmic
13	3 cycle logarithmic
14	4 cycle logarithmic
15	5 cycle logarithmic
16	6 cycle logarithmic

For example, the calling sequence words

```
$SG:4:2
```

will cause the following grid to be drawn:

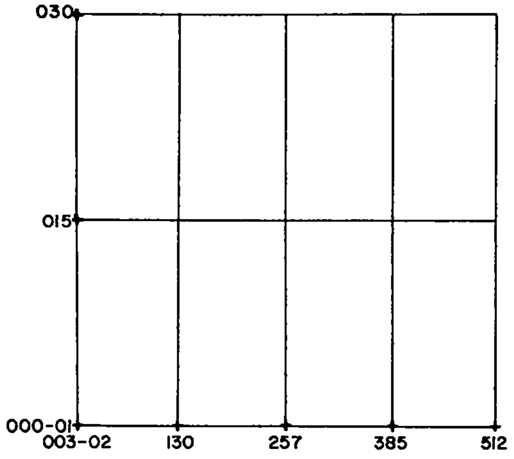


Any combination of options may be used, e.g., six cycle logarithmic ($P_1 = 16$) versus 50 linear intervals ($P_2 = 10$), etc. On logarithmic grids the main division lines are drawn heavier for easy readability.

3. Any grid produced by the preceding option can be labeled by using the "label grid" command <u>immediately</u> <u>following</u> the "select grid" command. Two calling sequence words are necessary, as follows:

```
$LG:XYBDS($WA)+P
```

where XYBDS($WA)+P contains the minimum X coordinate, the next location contains the maximum X coordinate, and the next two locations contain, respectively, the minimum and maximum Y coordinates. The labeling of a logarithmic scale follows the usual conventions with a "+" mark placed at each point labeled. Linear labeling consists of "+" marks placed along the left and lower border at points defined by the option and a 3 digit signed integer at each "+" mark with a 2 digit signed power of 10 added at the origin. For example, if we wish to label the grid of the previous section, and X_{min} = 3.15621-02, X_{max} = 5.1231, Y_{min} = 0, Y_{max} = 3, then the grid as labeled would look as follows:



Because of space limitations, the linear grid of 50 intervals is labeled only at 25 points.

4. The fourth option allows the writing of remarks either horizontally or vertically to describe the graph and the information contained in it. The format of the three calling sequence words necessary to do this is as follows:

$$\$WRN, REM(\$WP): R(\$WA)+P_1: C(\$WA)+P_2 .$$

where N = 0 means "write horizontally," 1 means "write vertically;" "REM" is the symbol for the remark to be written; and the next two quantities addressed are floating point numbers specifying the row and column at which printing is to begin. A film frame is considered to be divided into 64 rows and 128 columns; however, both R and C may be multiples of 1/4 to allow more exact positioning, superscripts, or subscripts. When writing vertically, the first character is positioned at row R and column C and succeeding characters are each spaced down one row. No more characters will be printed after row 64 is reached. When writing horizontally, the first character is positioned as before and succeeding characters are each spaced one column to the right. After column 128 is reached, the next character (if any) will be positioned one row down and at column 1. Note that the graph area defined previously is above row 57 and to the right of column 15.

5. To plot a series of points on a grid, the following five calling sequence words are necessary:

    ```
    $PFN,M:XC($WP):YC($WP):XYBDS($WA)+P:TEMP($WP)
    ```

    where N = 0 means "do not connect successive points with a line," N = 1 means "connect successive points;" "XC" is the block containing the ordinates (or the logarithms of the ordinates, if a log grid is used); "YC" is the block containing the abscissas (or the logarithms of the abscissas, if a log grid is used); the four words starting at the address "XYBDS($WA)+P" contain the minimum and maximum ordinate, and the minimum and maximum abscissas, in that order; and "TEMP" is a block the same length as "XC" and "YC" which "$MP" can use the temporary storage. "M" is the decimal equivalent of the character to be plotted which can be determined from the following table:

TABLE VIII

DECIMAL EQUIVALENTS OF PLOTTING CHARACTERS

CHAR.	DEC.	CHAR.	DEC.	CHAR.	DEC.	CHAR.	DEC.
blank	0	+	16	-	32	0	48
1	1	A	17	J	33	/	49
2	2	B	18	K	34	S	50
3	3	C	19	L	35	T	51
4	4	D	20	M	36	V	52
5	6	E	21	N	37	V	53
6	6	F	22	Ø	38	W	54
7	7	G	23	P	39	X	55

TABLE VIII (Continued)

CHAR.	DEC.	CHAR.	DEC.	CHAR.	DEC.	CHAR.	DEC.
8	8	H	24	Q	40	Y	56
9	9	I	25	R	41	Z	57
∂	10	π	26	. (plotting point)	42	0	58
=	11	. (period)	27	$	43	,	59
		)	28			(	60
"	12	β	29	*	44	∫	61
'	13	±	30	γ	45	Σ	62
δ	14	?	31	~	46	□	63
α	15			d	47		

If one desires to superimpose two characters, this can be done by setting M = 64 x (1st char.) + (2nd char.). The points are plotted with the character or characters specified, and successive points are connected by a vector, if desired. The <u>maximum</u> vector length allowed is 1/16 of the size of the film frame.

6. To generate a pair of axes through a given point, the following <u>three</u> calling sequence words are used:

```
$GA, XYBDS($WA)+P: XZERØ($WA)+P:YZERØ($WA)+P
```

where "XYBDS" is as described above, and "XZERØ" and "YZERØ" represent, respectively, the ordinate and abscissa of the point through which axes are to be drawn. The axes cover the 900X900 grid area only. As with "$PR," "$MP" detects certain errors, none of which are serious enough to cause it to return control to IVY, and prints appropriate diagnostic comments. Remedial action taken in the event of some errors is described in the table of error numbers. Some information on "$MP" diagnostics is included in Chapter 7, page 159.

J. <u>The disk program</u>. "$DK" is used to write blocks of data on the disk unit of the 7030, and on the disk units of 7090 machines which have them attached, and to read them back as needed. Although some 7090's do not possess a disk unit, "$DK" is nevertheless a valid subroutine on all 7090's, using a tape to simulate the disk. Up to 32 blocks of data, remarks, code, or calling sequences may be written on the disk.

1. To write a block on the disk, the following two calling sequence words are used:

   $WRN,ID($WA)+P: DATA($WP)

   where "N" (only on 7090's not having a disk unit) specifies the tape unit being used to simulate the disk. Only one tape unit may be used — i.e., "N" must always be the same number in a given program. On the 7030, and on 7090's having a disk, "N" is ignored. "ID(SWA)+P" addresses an "ID" word in the same manner as in the "$TP" calling sequence. This "ID" is entered into a 32-word table and is assigned an arc number on the disk (on some 7090's, a record number on the tape specified). The block "DATA" is then written in this arc (or record). If 32 blocks have already been written, an error indication is given unless the "ID" specified is the same as one of the previous twenty.

2. To read a block from the disk, one enters

   $RDN,ID($WA)+P: DATA($WP)

   The "ID" is sought in the table mentioned above, and its location on the disk (or tape) discovered, and the record is read into the block specified by "DATA." If the "ID" cannot be found in the table, "$DK" gives an error indication and control returns to IVY. (The absence of the "ID" from the table indicates that a record with this "ID" was never written.)

K. The instructions to operator routine. "$ØP" functions similarly to the "Ø" card in that it prints a remark on-line, sounds a gong if possible, and halts or waits until the "start" (7090) or "console signal" (7030) is pressed, signifying that the instructions have been carried out. The calling sequence to "$ØP" is as follows:

($P,$ØP: REMARK($WP)),...

where "REMARK" is the name of the remark to be printed as a comment to the operator.

L. The character manipulation program. "$CM" is used to change remarks or format statements or to construct new ones under programmer control. "$CM" may not be used to alter calling sequences on "K" cards (see Chapter 8, pages 182-184). A number of calling sequence words cause the various necessary functions to be carried out.

1. To set an entire block to contain the same character, e.g., blank, zero, etc., the following calling sequence word is used:

   $SN, REMARK($W)

   where "N" is the two-digit decimal equivalent of the character desired (see Table VIII), and "REMARK" is the block to be set.

2. To move from one to fifteen characters from one remark block to another, the following five words are used:

   $MN:AD($WA)+P_1:REM1($WP):AE($WA)+P_2:REM2($WP)

   where "N" is a hexadecimal digit, $1 \leq N \leq F$, specifying the number of characters to be moved; location "AD($WA)+$P_1$" contains a fixed point number specifying the character number in remark block "REM1" where the "N" characters to be moved begin; and "AE($WA)+$P_2$" contains a fixed point number specifying the character number in remark block "REM2" where the "N" characters moved will begin.

3. To compare from one to fifteen characters in one remark block to the same number of characters in another remark block, the following five words are necessary:

   $CN:AD($WA)+P_1:REM1($WP):AE(SWA)+$P_2$:REM2($WP) where again $1 \leq N \leq F$ specifies the number of characters to be compared; "AD($WA)+P$_1$" contains a fixed point number specifying where the "N" characters to be compared begin in "REM1;" and "AE($WP)+P$_2$" contains a fixed point number specifying where the "N" characters in "REM2" begin. The specified "N" characters in "REM1" are compared to the "N" characters in "REM2;" if equal, "$CS1" is set to 1; if not equal, "$CS1" is set to zero. To clarify the usage of "$CM," a number of examples are included below.

EXAMPLES:

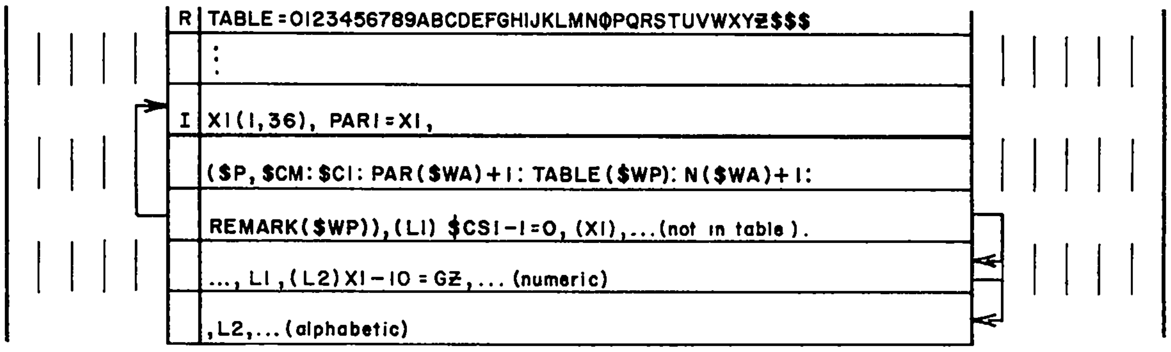
1. To alter a format statement: If "R1" and "F1" are as shown in the illustration below, then the calling sequence illustrated will change the "P" to a "T" and the "7" to a "3" in the format statement, thus altering it to print numbers with less significant digits off-line:

R	FI = CRP, 1.1.1.7.2, $ $ $
R	RI = T3 P7 $$$
	:
I	PARI=3, PAR 2=3, PAR3=2, PAR4=11,
	($P, $CM: $MI: PAR($WA)+1: RI ($WP): PAR($WA)+2:
	FI ($WP): $MI: PAR($WA)+3: RI($WP): PAR($WA)+4:
	FI($WP)),...

2. Defining a block and setting it to blanks: If remark block "R2" is defined as shown below, space is allotted for it but is not cleared in any way. To ensure that the block contains only legal characters (in this case, all blanks), the following is used:

R	R2 (759) = $$$
	:
I	($P, $CM: $SOO, R2($WP)),...

3. Determination of a character's position in a table. In the following example, we wish to see whether a character of "REMARK" lies in "TABLE" (i.e., whether it is alphanumeric) and if so, its position, so that we can determine whether or not it is alphabetic:



```
R | TABLE=0123456789ABCDEFGHIJKLMNØPQRSTUVWXYƵ$$$
  | :
I | X1(1,36), PAR1=X1,
  | ($P,$CM:$C1:PAR($WA)+1:TABLE($WP):N($WA)+1:
  | REMARK($WP)),(L1) $CS1-1=0, (X1),...(not in table).
  | ..., L1,(L2)X1-10=GƵ,... (numeric)
  | ,L2,...(alphabetic)
```

The calling sequence is entered once for each value of X1, from 1 to 36. Each time PAR1, the character position in "TABLE," is set to the value of X1. It is assumed that N1 contains the character position in "REMARK." If at some given exit, "$CS1" contains 1, the character is indeed in "TABLE," and the contents of X1 are tested to see whether the character is numeric (X1 $\geq$ 10) or alphabetic (X1 > 10). If the index loop is completed without $CS1 being set to 1, the character is not in "TABLE" and hence not alphanumeric.

M. Summary. In this chapter we have dealt with the calling sequences to all internal IVY subroutines which are available to the programmer. With these subroutines one can do a wide variety of input-output operations, indicator and switch testing, and internal manipulation. Our study of the IVY "$"symbols is now complete. In Chapter 9 one can find a table of all the IVY "$" symbols (page 187), as well as quick reference tables for calling sequences to the various subroutines described in this chapter (pages 197-202).

CHAPTER 7

IVY ERROR INDICATIONS

Each time any of the routines in IVY detects an error, an error diagnostic is printed. Each routine prints particular information, described below; all these printouts have in common an octal number in columns 73-78. This number, called the error number, can be looked up in a table of error numbers published separately from this manual, which is available to manual holders and at the console of each machine on which IVY is run. This table, arranged in numerical order, gives a complete description of the error which caused the diagnostic printout. The table is not included here both because of its length and because from time to time additions to, corrections of, and deletions from the table will take place. The details of the rest of the error printout for each routine are described in this chapter in a brief manner so that the other information in a diagnostic line can be easily interpreted.

A. "$LD" prints as follows:

1. Columns 1-72: contents of the card on which the error was detected, if possible. If these columns are blank, either the card is the same as the one on which a previous error occurred, or else the card contents are not available to "$LD." In the latter case the type of

error indicated, and the other information printed on the line, are usually sufficient to localize the error.

2. Columns 73-78, the error number described above.

3. Columns 79-84, an alphanumeric symbol to help locate the error on the card printed in columns 1-72. In some cases, this is the symbol for the formula set or formula in which the error occurred. In others this is the last data symbol encountered on the card before the error was detected.

4. Columns 85-120, other information, where possible, to help localize the error. For instance, if "L" cards are being loaded, columns 85-90 contain the mnemonic code for the operation which caused the error indication. In all cases, the explanation of the octal error number describes the information which appears.

When data or remarks are being loaded, any error in the definition of a block or in its loading causes the control word to be flagged. When code is being loaded or assembled, errors caused by referring to such erroneous data are caught, and the guilty instructions are replaced by transfers to return control to IVY. The same procedure is followed in the case of errors peculiar to the code itself. When calling sequence blocks are being loaded, an error of some sort in a calling sequence word entry causes the erroneous word to be replaced by a word of all ones, which, when detected by the subroutine using the calling sequence, causes an error indication, and in some cases, causes control to be given to IVY (e.g., in"$TP," "$AP," etc.).

B. The form of the error diagnostic for "$AP" is similar to that for "$LD," except that in many cases the card image is not available to be printed in columns 1-72. This program makes up for these difficulties by printing additional information in the remaining columns of the diagnostic.

C. "$TP" and "$DK" print diagnostics as follows:

1. Columns 1-72 contain, when possible, the ID of the record being written in the form of an octal number prefixed by "ID=," and the tape number, i.e., the IVY number as well as the channel and unit number on the machine being used.

2. Columns 73-78, as usual, contain the octal error number.

3. Columns 79-84 contain the alphanumeric portion of the "$XXX" part of last calling sequence word encountered before the error was detected.

D. "$ØD" prints only an octal error number and the symbol of the last block dumped before the error was detected.

E. "$PR" and "$PH" print the octal error number in columns 73-78, and usually a symbol in columns 79-84. This symbol may be the name of an erroneous format statement or the last block encountered in the calling sequence. In some cases it represents the alphanumeric part of an illegal "$XXX" calling sequence command. Which of these it represents is always explained in the error number description.

F. "$MP" and "$CM" also print merely the octal error number and a symbol which may represent the alphanumeric part of a "$XXX" calling sequence entry or the name of a block. The error description table explains how this symbol should be interpreted.

Other than the detailed description of the interpretation of error numbers, little else can be said about error diagnostics. The above information, plus what has been said about the error detection procedures of the various subroutines described in Chapter 6, should suffice to enlighten the programmer on exactly what error he has committed when this information is combined with the error number tables.

In general it has been found that a great variety of rather specific errors can be spotlighted by this method. If,instead, a line containing a comment describing the error were printed, each comment would take up considerable space in core and hence errors noted would be fewer and more general. Although the IVY number system presents more work for the coder, until he learns the more common error numbers and their meanings, it nevertheless makes the location of errors (often rather difficult in some other coding systems) a fairly trivial matter. The further advantage of IVY, that a code containing errors may be executed until an erroneous instruction is encountered, greatly simplifies the process of debugging and, indeed, is an ideal system for the class of fallible coders which is an extremely large subset of the class of all coders. And yet, error checking is in general a simple process which does not take a large fraction of time away from the IVY program so that coders with debugged decks may also run with advantage in the IVY system.

It should be noted that IVY by no means is able to detect all possible errors because of space and time limitations. More subtle errors, and particularly, coding errors involving e.g., index loops which have large enough limits to destroy parts of the code and other errors in method, can be detected and corrected only by a human being. Thus, the complete absence of error diagnostics is no guarantee that a code is debugged. Only the successful running of a code to completion with the corresponding production of correct results is a guarantee of this; and, indeed, such a run guarantees only that the part of the code executed is debugged.

A further note of caution is now appropriate: although IVY may not detect a certain error, in many cases the undetected error will cause an error indication to be generated on some subsequent instruction which appears to be correct. Detectable errors may also produce this phenomenon. In a case like this it is advisable to examine the place at which the <u>generated</u> error printout occurred and work backwards from this point, both sequentially and in a non-sequential fashion to the data blocks and other locations referred to by the instruction on which the diagnostic occurs. In this manner the true source of generated error diagnostics can usually be found.

CHAPTER 8

CODING EXAMPLES

<u>Coding efficiently</u>. Coding in an efficient manner is largely an acquired characteristic and cannot really be taught except by experience. Nevertheless, a few tips can usually be given on virtually any system which will accelerate this learning process. Hence the first section of this chapter is dedicated to coding efficiency. An <u>efficient code</u> is defined as a code which, when translated to machine language, optimizes a number of quantities: the number of machine instructions which should be minimized; the execution time of the code which should again be a minimum; the ability of a code to produce accurate results which should be as great as possible; and the conciseness of the code which should be as great as possible, both to facilitate understanding of it by persons other than the original coder and to simplify debugging. Thus we see that constructing an efficient code presents a rather complex minimax problem the solution of which is by no means trivial. Nevertheless, it can be approached rather closely by the experienced coder; the following considerations should aid considerably in producing an efficient code.

1. Minimizing the number of machine instructions. To do this, a programmer must for the moment forget that he is human and try to think as a machine. A few simple rules will illustrate this:

a. Although IVY allows for the use of an almost unlimited number of index registers, if more are specified than a particular machine contains (three on the 7090 and fifteen on the 7030), a corresponding increase in the length of the code is required to simulate the extra index registers. If, however, one does need more index registers than are provided physically, it is more efficient to allow for this number on the "S" card and let IVY handle the problem than to specify, e.g., three, and do all the necessary storing and restoring in the algebraic language.

b. If an algebraic expression occurs as a part of two or more larger statements, it is advisable to compute this expression first in terms of some symbol and use the symbol, rather than the expression, in the larger statements of which it is a part. For example, the following is an example of an inefficient code:

```
I|A1.2=C($W)+CX(I)+CXX(M), A2.2=C($W)+CX(I)+CXX(M)-CX(K)+CX(J),
```

Coded efficiently, this example appears as follows:

```
I|T1=C($W)+CX(I)+CXX(M), A1.2=T1,A2.2=T1-CX(K)+CX(J),
```

where "T1" is some suitably chosen temporary location.

c. In the case where a quotient contains a complicated expression in the denominator, a person using a hand calculator will usually compute the denominator first, write it down, compute the numerator, then while the result is still on the dials, divide by the denominator. This is more efficient than computing the numerator first since in the latter case one must write down <u>two</u> numbers. Electronic computers work in the same way. Therefore IVY provides the reciprocal divide operation "//," which should be used whenever a relatively complicated expression appears in the denominator. For instance, some <u>inefficient</u> expressions are

```
I|G=C+B/(D*E+F), H=B+C/(D+E)+F,
```

<u>efficient</u> expressions for the same things are:

```
I|G=D*E+F//C+B, H=D+E//(B+C)+F
```

d. To generalize a bit on the above, two expressions can often be written with different arrangements of terms and parentheses. In general the one with fewer parentheses is <u>always</u> more efficient than the other. Most often, each set of parentheses in an IVY expression causes a "store" instruction to be generated. Hence, the fewer parentheses an expression contains, the shorter the machine language code. For example:

```
inefficient:    -C2+((C2**2-(4.0*C1*C3))·$R)/(2.0*C1)
efficient:      2.0*C1//C2**2-(4.0*C1*C3)·$R-C2
```

e. Another point to consider under this heading is the relation between storage used for code and storage used for data. Generally, if one is decreased the other increases, but not necessarily in direct proportion. For example, in using the "ØJ" convention to skip the assembly of one of two alternative formulas (page 104), giving up one location for a parameter to control this can save many locations that the alternative formula would occupy if assembled. And, on the other hand, one can often save a considerable number of data locations by using double-stored data. The code will be lengthened, of course, by the use of instructions which refer only to the "Q" or to the "T" portions of these numbers, thus producing what one might call a less efficient code; but the increase in the length of the code will usually not be as great as the decrease in the length of the data, and if the amount of storage used rather than the speed of the code is the paramount consideration, then the use of double-stored data is definitely advisable.

2. Minimizing the running time of a code. This is a very elusive technique to develop, and there are two rather contradictory methods for doing it. One is to reduce the length of the program using the techniques described in 1 above. The other is to increase the length of the code by avoiding index loops and subroutines as much as possible. It is usually best to compromise on the latter method, since adding index loops and subroutines increase the efficiency of the code in other ways by making it shorter and by making it easier to read. No really hard and fast rules can be given here — the programmer himself must consider his own peculiar requirements of length, speed, simplicity, etc., to reach a decision. Only a very short and simple code can be truly optimized one hundred percent with respect to speed.

3. Producing accurate results. Results can be affected not only by cumulative errors resulting from roundoff and truncation, but also by errors resulting from the inherent accuracy of the method used. The latter can be obtained from any good text on numerical analysis. About roundoff and truncation errors, however, little is known, but a few simple rules can be stated which help to minimize errors from this source.

a. When adding together a table of floating point numbers, the smaller numbers should be added in first. The cumulative effect of these numbers may change the result of the calculation, but might pass unnoticed if the larger numbers are first combined to produce a number compared to which each of the smaller numbers taken alone is not significant.

b. Along the same lines, when any assortment of numbers is being combined arithmetically, it is best to combine numbers of approximately the same order of magnitude before working up to higher order results. This point is made more clear by a consideration of the evaluation of a polynomial, e.g., of fourth degree:

```
inefficient:    C5*(Y**4)+(C4*(Y**3))+(C3*(Y**Y2)+(C2*Y)+C1
efficient:      C5*Y+C4*Y+C3*Y+C3*Y+C2*YC1
```

In this case, the number of parentheses also indicates the relative ineffiency of the first example. However, in the following example (in which a number is raised to the 5/2 power) there are no parentheses to give a clue. The efficient way of writing the operation, however, contains fewer operations:

```
     inefficient:      AX.$R**5 or AX**5.$R
     efficient:        AX.$R*AX*AX
```

The following example can be given to illustrate an efficient way of computing the difference of the squares of two numbers:

```
     inefficient:      Y**2-(Z**2)
     efficient:        Y-Z*(Y+Z)
```

The latter will give greater accuracy because it follows the rule of multiplying the two numbers of roughly the same order of magnitude, whereas in the first expression the two numbers may differ considerably more in order of magnitude. Thus, to compute, for instance, the expression $AX^{5/2} - BX^{5/2}$, we use the following sequence:

```
     I|T1=AX.$R.$R*AX, T2=BX.$R.$R*BX, RESULT=T1-T2*(T1+T2),
```

Although a great variety of examples can be paraded forth to illustrate this type of efficiency, it is hoped that the above will be sufficient.

4. Conciseness of the code. An extremely concise code is usually not efficient from the standpoint of speed but a concise code has other advantages such as readability, aesthetic beauty, and — especially important for some coders — compactness. Furthermore, a concise code is often easier to debug than one which sprawls out like some sort of enormous octopus. The following rules should be observed to a greater or lesser degree depending on the conciseness desired:

a. Wherever possible, loops should be used to evaluate expressions where repetitious operations occur. Although loops are slower in execution than linear sequences, they produce a beautifully compact machine language code. For instance, consider the evaluation of a polynomial of degree N:

```
inefficient:  I|RESULT=C(N+1)*Y+C(N)*Y+...+C(3)*YC+(2)*Y+C(1),...
efficient:    I|RESULT=C(N+1),X1(N,1),RESULT=SM*Y+C(X1),(X1),...
```

b. The same sequence of instructions, even if operating on different quantities, need never appear twice. Instead, the calculation being performed can be coded as a subroutine, entered from any point in the code where this calculation is desired. By subroutinizing a code so that each part of it has a separate function to perform, debugging is greatly simplified, too, since abnormalities in the results can be quickly traced to the guilty subroutine. The formula set-formula-local entry structure of IVY makes it singularly trivial, and almost compulsory, to code in this manner.

The use of index multiples. (For review, the reader is referred to Chapter 3, page 54, where index multiples are first discussed.) Suppose, for example, that an array "G" and its index multiples "GX" and "GXX" are defined as follows:

```
D|G(AT, BT, CT),GX(BT)=(M)AT,GXX(CT)=(M)AT*BT,...
```

Suppose, now, that we wish to compute a quantity "D" involving the element of "G" having indices (3, 2, 6). Then either of the following sequences of instructions can be used:

```
I|X1=3+GX(2)+GXX(6), D=G(X1)+...
```

or

```
I|A1.1=G($WP+3+GX(2)+GXX(6), D=G(A1)+...
```

In either case, the address given for the element of "G" is G($WA)+3+(1*AT)+(5*BT) which corresponds to the general formula in Chapter 3, page 54.

If, now, we wish to address a general element (i, j, k) of the array "G," we can do so in any of the following ways. (Let us assume that i is in X1, j is in X2, and k is in X3):

```
1.  I|X4=X1+GX(X2)+GXX(X3), D=G(X4)+...
2.  I|A1.1=G($W)+X1+GX(X2)+GXX(X3), D=G(A1)+...
3.  I|A1.1=G($W)+GX(X2)+GXX(X3), D=G(X1+A1)+...
```

and so on. As can be seen, there are a great variety of ways in which a general element of an array can be addressed. Each is best for its own particular purpose. The decision of which of the above to use, or whether to use another method, is up to the programmer. For example, we recall that arrays such as "G" are stored column-wise, i.e., the first index varies most rapidly. Method 3 can be used when we wish to operate on the elements of a particular column or columns of an array in order, indexed by X1. For instance, the example in Chapter 4, page 78, illustrates a method of operating on the two-dimensional matrices which make up the three-dimensional block "C."

<u>Index loops</u>. We encountered our first example of an index loop in Chapter 4, page 92. There the index ran between two values, one of which

was 1 and the other of which was a parameter algebra expression, thus:

$X_n(1,P),\ldots$ or $X_n(P,1),\ldots$

$\vdots$

$(X_n),\ldots$ $(X_n),\ldots$

If one desires to set up an index loop between parameter limits one of which is not 1, and/or increment the index by values other than ±1, the form for doing this as we learned in Chapter 5, page 99, is as follows:

$$X_n=N_1,L_m,\ldots$$
$$\vdots$$
$$X_n=X_n+N_2,\ (L_m)X_n-N_3=C,\ldots$$

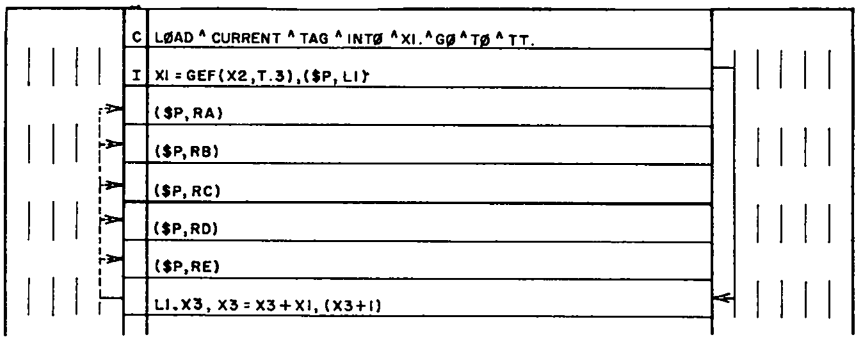
where N_1 is the initial value of X_n, N_2 is the increment (positive or negative), and N_3 is the final value of X_n. "C" represents some appropriate condition for the branch. We note that the N_i must be parameters since the dynamic modifier "A" is not present. An example of this type of index loop is given in the next section of this chapter, page 172.

To construct an index loop in which any of the N_i as shown above are computed or dynamic quantities, the modifier ".A" must occur after the index register symbol left of the equal sign. For example, if N_1, N_2, and N_3 are all computed quantities, the following is used:

$$X_n.A=N_1,L_m,\ldots$$
$$\vdots$$
$$X_n.A=X_n+N_2,\ (L_m,A)X_n-N_3=C,\ldots$$

Thus, with the use, where necessary, of the "L_n" entry and the ".A" modifier, complete flexibility can be attained in writing index loops.

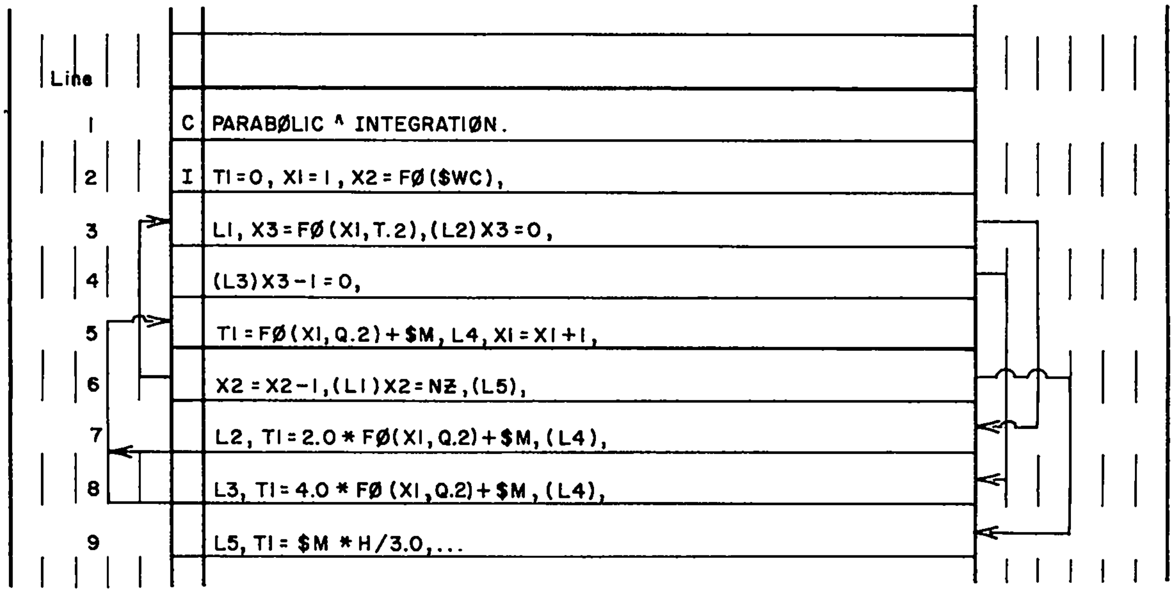
Double Stored numbers. The entry of double-stored numbers on "E" cards, and some motivations for their use, e.g., to use the tags to specify boundary and interior points, etc., have been discussed in Chapter 3, pages 58-59. If it is a simple case of distinguishing between boundary and interior points, only a one-bit tag is needed. If, however, the boundaries have different conditions on them, more tag bits are needed. Again, the choice of how many tag bits to use in a double stored block is the programmer's. Of course, the fewer tag bits used, the greater the accuracy of the high-order or "Q" portion of the number. One should recall that the length of the tag can be minimized by considering that only four tag bits, for instance, are needed to specify fifteen conditions, and not fifteen bits. Where a number of different alternatives are specified by the tag bits, one can load the contents of the tag into an index register and go to a transfer table which carries control to the various alternative routines, as shown in the following example (where the tag length is 3, but the tags take on only the five values 0, 1,..., 4):



```
C LØAD ^ CURRENT ^ TAG ^ INTØ ^XI.^GØ ^ TØ ^ TT.
I XI = GEF(X2,T.3),($P,LI)
  ($P,RA)
  ($P,RB)
  ($P,RC)
  ($P,RD)
  ($P,RE)
  LI.X3, X3 = X3+XI, (X3+I)
```

The next example illustrates a parabolic rule integration (a special case of Simpson's rule for equal intervals of the independent variable). Here it is assumed that the block "FØ" contains N+1 values of the function f(x) to be integrated in order of increasing x. N is an even integer and the interval in x between values of the function f(x) is 2h where h is contained in the location represented by the symbol "H." We further assume that the block "FØ" is double-stored with a tag length of two bits. f(x) and $f(x_N)$ have tags of 2, $f(x_{2i})$ have tags of zero, and $f(x_{2i+1})$ have tags of 1. The integration, the result of which is placed in "T1," proceeds as illustrated. Recall that the formula for this type of integration is

$$I = \frac{h}{3}\,(f_1+4f_2+2f_3+\ldots+4f_N+f_{N+1}).$$



```
Line
 1   C  PARABØLIC ^ INTEGRATIØN.
 2   I  T1=0, X1=1, X2=FØ($WC),
 3      L1, X3=FØ(X1,T.2),(L2)X3=0,
 4      (L3)X3-1=0,
 5      T1=FØ(X1,Q.2)+$M, L4, X1=X1+1,
 6      X2=X2-1,(L1)X2=NZ,(L5),
 7      L2, T1=2.0*FØ(X1,Q.2)+$M,(L4),
 8      L3, T1=4.0*FØ(X1,Q.2)+$M,(L4),
 9      L5, T1=$M*H/3.0,...
```

Note the use of "L1" to simulate an index loop. This technique, described in the previous section, is best used when an index register runs between computed values or values for which there exist no symbols. If, of course, we have a parameter M which gives the length of the "FØ" block, we can replace the last expression in line 2 with "X2(M,1)," and the first <u>two</u> expressions in line 6 with "(X2),".

<u>A complete IVY code</u>. The reader is now referred to the complete code illustrated on page 4 which is our next subject for discussion. This program, although short and simple, illustrates principles which are followed in any IVY code, regardless of length or complexity, as well as demonstrating some further coding techniques.

First of all, the gross organization of an IVY code is illustrated in the example. This organization is as follows:

1. "*" card to identify output. In this case off-line output is not produced, but the contents of the "*" card will be printed on-line to identify the printout. This is also useful for logging purposes.

2. "S" card to initialize program. Note that no formulas occur in either formula set and that there are no store addresses. Four index registers and three local entries are used, and two numbered symbols "R" are specified.

3. "D" cards. In this example we both define and load the two vectors "AØ" and "BØ," name the two formula sets "FLOW" and "SUBR," and set aside one word for temporary, called "T."

4. "R" cards, which load two remarks: a format statement and a comment. The error comment is not really necessary in this case since the vectors obviously have equal counts. It is entered for illustrative purposes.

5. If any "K" cards were entered, they would occur next in sequence.

6. "A" card to write the first formula set on tape 2, file 1.

7. "I" cards entering the first formula set, called "FLØW."

8. An "A" card to write the second formula set on tape 2, file 2.

9. "I" cards entering the second formula set, called "SUBR."

10. "A" card to read in and assemble the first formula set. Note that an "X" card is not necessary since "FLØW" ends with an execute statement.

11. If more data were to be loaded, "E" cards and "X" cards would occur after the last "A" card.

Let us now consider the code in more detail. We note that each formula set is preceded by an "A" card to place it on tape 2. In a code this short, both formula sets can be placed in the same file, if desired, by using only one "A" card in front of "FLØW," and moving the execute statement on line 13 down to the end of "SUBR," on line 22. However, the procedure shown here is equally valid, and is especially useful for longer codes.

In all IVY codes, long as well as short, there should be a flow code as in this one; hence the choice of the name "FLØW." The flow code is a formula set, the function of which is to control the loading of data and the conversion of other formula sets, as well as the access to and from the other formula sets of the program which should be semi-independent entities. Thus we note, as in the simple example, that the flow code consists mainly of branches and calling sequences to subroutines, supplied either by the coder or by IVY. Longer programs will of course have longer and more detailed flow codes constructed from a number of formulas, which test various parameters and branch to various formula sets depending on the values of these parameters. Thus the flow code acts as a sort of traffic policeman, directing the flow of control along the paths desired in a particular run.

In the case of a code of such length that the entire program will not fit into memory at one time, the flow code can also assume the role of monitor, directing the allocation of memory space, altering control words, and reading new formula sets into the space formerly occupied by others whose purpose is completed. Coding examples involving this particular technique occur in Appendix 1, since they are of little interest to most coders.

In the flow code of our example, the assembly program "$\$$AP" is first entered to convert the subroutine "SUBR" into machine language. Control is then returned to "FLØW" which executes a pathfinder branch to "SUBR." Note that at the time "FLØW" is converted, this pathfinder branch has not yet been assigned an address since "SUBR" is not yet converted. When "FLØW" goes to "$\$$AP," however, this address is computed and inserted. Thus it is perfectly legal for a code to contain branches to uncoverted routines as long as the routines are converted before branch references to them are executed as is true in this case.

The calling sequence to "SUBR" contains the control words of two vectors "AØ" and "BØ" of which the dot product, i.e., the sum of the products of corresponding elements is desired, and the address of a location "T1" where the result is to be stored. The error return contains, as required, a pathfinder branch to a location which prints out an error comment. If control returns to the normal return, the result is printed on-line. In either case, at the end of the program, control is returned to IVY by the pathfinder branch

,($P,$LD),

which, we recall, must always be the last executed instruction in any IVY code. The loading program selects the reader or off-line tape in an attempt to find the next IVY code, and if none exists, a halt occurs.

The subroutine first causes the pathfinder contents to be placed in X4, then sets up a temporary block "$D" four words long. Index registers are stored in the first three, and the dot product is accumulated in "$D4," from where it is later transferred to "T1." The information in the calling sequence is examined and if found in error, the contents of X4 are decreased by one so that the exit is modified to cause control to return to the error return. Otherwise, the computation is completed and control returns normally.

More discussion of IVY calling sequences. In Chapter 6 we considered in detail the calling sequences to IVY internal subroutines. However, this section is designed primarily to consider subroutines which the programmer has coded and the techniques of constructing and handling these programmer calling sequences. This discussion supplements in more detail what has been said in Chapter 5, pages 105-110, on calling sequences.

Usually the "$WA" portion of a calling sequence word contains an address or count, and this address or count is best used in an index register for address modification or for counting purposes. One may observe both of these in the illustrative example discussed in the previous section. Either of the entries

:SYMBØL($W): or :SYMBØL($WA):

causes the control word address to be placed in the "$WA" portion of the calling sequence word; it should be recalled that the control word address is one less than the base address of the block. Thus, in line 18 of the illustration, we see the two expressions "$Z(X2+1)" and "$Z(X3+1)" occurring. We also see on line 10, the entry "T($WA)+1" in the calling sequence which gives the true address of "T1," and hence this is referred to in line 20 as "$Z(X1)". To load the "$WA" portion of a calling sequence word into an index register "X_m" if "X_n" contains the pathfinder contents, either of the following entires may be used:

$$X_m = \$Z(X_n+N), \text{ or } X_m = \$Z(X_n+N, \$WA).$$

In other words, the index register is normally loaded from the "$WA" portion if nothing else is specified. However, the latter entry is recommended to avoid confusion.

A parameter, literal, or the result of a parameter algebra expression may also be entered in the "$WA" portion of a calling sequence word by one of the entires

:SYMBØL: or :N: or :P:

where "N" represents a fixed point literal and "P" represents a parameter algebra expression. This number may be used in arithmetic or loaded into an index register if it is less than 2^{15} on the 7090, 2^{18} on the 7030. If loaded into an index register, again the "$WA" modifier is recommended though not necessary. If used in arithmetic, and if a "$XXX" entry occurs in the same calling sequence word, the "$WA" modifier <u>must</u> be used to mask out the "$XXX" portion of the word.

A quantity can be placed in the "$WC" portion of a calling sequence word by either of the entries

:SYMBØL($W): or :$XXX:

where "XXX" represents one, two, or three <u>alphanumeric</u> characters. In the first entry, the control word count of "SYMBØL" appears in the "SWC" portion of the calling sequence word, and in general this will be loaded into an index register for counting or testing purposes by means of the entry

$$X_m = \$Z(X_n+N, \$WC),$$

as we see in line 16 of the illustration. The "$WC" modifier <u>must</u> be present. If, however, a "$XXX" entry occurs, it can be loaded into an index register on the 7090 only if there are one or two characters since three characters occupy 18 bits. In most cases two characters are perfectly sufficient for the coder's purposes so this can be done with impunity. The X's can then be tested for control purposes by using conditional branches dependent on dynamic index arithmetic expressions. For example, if the entry in calling sequence word 5 consists of only one character, we might test whether this character is "S" by the following:

```
X1 = $Z(X4+5, $WC), (L1)X1-50 = 0,...
```

The decimal representation of "X" is obtained from Table 9.1, Chapter 9, page 186.

If the "$XXX" entry contains three characters, it is best to test its value using fixed point arithmetic. For example, if we wish to test the fourth entry in a calling sequence for "ABC," the following should be used:

```
(L1)$Z(X4+4,$WC)-17*64*64-18*64-19 = 0
or simply (and more efficiently)
(L1)$Z(X4+4,$WC)-70803 = 0,
```

where again the decimal representations of "A," "B," and "C" are obtained from Table 9.1.

Constructing a variable length calling sequence. A calling sequence which always contains a fixed number of words such as that illustrated presents no particular problems. The subroutine is simply coded to take this length into account, and always returns control to an exit or exits immediately following the calling sequence. However, if the programmer wishes to code a subroutine the length of whose calling sequence varies from one entry to another, a number of problems arise, and these are discussed in this section.

1. How to tell when the end of the calling sequence has been

reached:

a. The easiest method of doing this is to place the calling sequences on "K" cards, and to set up pathfinder branches to the subroutine in this fashion:

(∲P, SUBRTN: KNAME(∲W)),...

where "KNAME" is the symbol of the calling sequence block which contains the particular calling sequence desired for this block. Then the subroutine, in outline, will look as follows:

```
I | SUBRTN: X1, X2 = $Z (X1+1, $WC), X3 = $Z (X1+1, $WA),
  | L1, ...
  | X2 = X2-1, X3 = X3+1, (L1) X2 = NZ, ... (exit)
```

The count and address of the "K" block are loaded into separate index registers, and the current calling sequence word can be addressed by "∲X(X3+1)" with "∲WA" and "∲WC" modifiers, if needed. At the end of the routine the count is decreased by 1, the address is increased by 1, and control is returned to the start of the subroutine if the count is not zero; otherwise, an exit is performed. Thus, we see that putting calling sequence on "K" cards allows us to detect the end of a variable length calling sequence by using the count of the block.

b. If one does not wish to use "K" blocks, the calling sequences can be included in the code, thus:

$$(\not{\$}P,SUBRTN:CSW_1:CSW_2:\ldots:CSW_N),\ldots$$

where each "CSW_i" is some calling sequence word entry. The problem of detecting the end of the calling sequence then becomes more acute. However, the problem can be solved in one of the following ways:

(1). The first calling sequence word can contain a parameter which tells the number of remaining calling sequence words.

(2). The <u>last</u> calling sequence word can differ from all other calling sequence words in an easily detectable manner, e.g., by containing a special "$\not{\$}$XXX" different from all others. A zero word is easily detectable, and can be entered by using

:$\not{\$}$,0:

as the last word in the calling sequence.

2. How to move forward through a calling sequence: If an index register is loaded with the location of the first calling sequence word, either from the pathfinder or from the control word of a "K" block, one can advance forward through the calling sequence either by advancing the index register by one each time a new calling sequence word is desired, or by using the contents of the index register to compute a stored address which is incremented by 1 each time a new word is desired, e.g., by the following technique:

$$L1,A1.n = X_m+1,$$

$$\vdots$$

$$X_m = X_m+1,(L1),\ldots$$

where X_m contains initially the location of the first calling sequence word, minus one, and "n" is the number of times the stored address "A1" is used.

Constructing a calling sequence block. The final topic in this chapter will be a consideration of the construction of calling sequence blocks. It should be recalled from Chapter 3, page 64, that calling sequence blocks entered on "K" cards can be assigned without being loaded with information; or such blocks may be only partially loaded with information. The programmer can then construct these blocks as he wishes, using only IVY algebraic instructions. Suppose the name of such a block is "KNAME," and we wish to construct a calling sequence word in the Nth position of this block. Then we can do this in any one of a variety of ways, for example:

KNAME(N,M) = expression,

X_i = N,KNAME(X_i,M) = expression,

X_i = KNAME($WA),$Z(X_i+N,M) = expression,

and so on. "M" is one of the modifiers "$WA" or "$WC," or may be omitted in some cases. The following examples will serve to illuminate the reader on the techniques involved:

1. To insert a "$XXX" entry: Suppose we wish to enter the expression "$ABC" in calling sequence word N. This can be done as follows:

KNAME(N,$WC) = 17*64*64 + (18*64)+19,

or more simply,

KNAME(N,$WC) = 70803,

where the decimal representations of the characters A, B, and C are obtained from Table 9.1, page 186. Note that the "$WC" modifier is <u>required</u> here.

2. To insert the control word of a symbol:

KNAME(N) = SYMBØL($W),

where no modifier is required on the left. Similarly, one may enter a control word modified by a parameter algebra expression as follows:

KNAME(N) = SYMBØL($W) + P,

3. To insert the control word address or control word count of a symbol in the "$WA" portion:

KNAME(N,$WA) = SYMBOL($WA),

or

KNAME(N,$WA) = SYMBØL($WC),

These expressions may also be modified by parameter algebra if desired.

4. To <u>modify</u> either the "$WA" or "$WC" portion of an already existing calling sequence word by some expression:

KNAME(N,$WA) = $M+P

or

KNAME(N+$WC) = $M+P

and so on. In general, by the use of such expressions, one can produce considerably more calling sequence words than by actually entering the word on the "K" card. Such unconventional calling sequence words are, of course, permissible as long as they do not occur in the calling sequences to any of the IVY "$" routines. Examples of entries which produce atypical calling sequence words are as follows:

```
KNAME(N,$WC) = SYMBØL ($WA),
KNAME(N,$WA) = SYMBØL,($WA)+SYMBØL2($WC),
KNAME(N)     = 3.1415926535*RADIUS
```

and so on.

It should be emphasized that calling sequences cannot be modified by using "$CM," the character manipulation routine. Calling sequence blocks are entirely different in form and in contents from remark blocks. Thus only the techniques discussed above can be used to modify these blocks.

Conclusion. These few coding examples should prove sufficient to get the IVY programmer off to a good start. Many other techniques can be developed with experience, and in general it is easier to do this with IVY than with most other programming systems. IVY combines the advantage of being an ideal system both for the beginning coder and for the virtuoso: for the former because it is a simple system in which it is almost impossible not to code properly; for the latter because of the abundance of sophisticated techniques which are available.

CHAPTER 9

SUMMARY AND TABLES

This chapter is intended to be a thumbnail sketch of the IVY manual. In it are gathered tables and summaries of the information covered in the first eight chapters for quick reference purposes. Nothing occurs here that has not occurred previously, but virtually any topic in the manual can be found in this chapter quickly and easily; page references to complete, more detailed discussion are always given.

TABLE 9.1

THE IVY CHARACTER SET, CARD PUNCHES, INTERNAL REPRESENTATION

CARD CHAR.	PUNCH	OCT.	DEC.	4020 CHAR.	CARD CHAR.	PUNCH	OCT.	DEC.	4020 CHAR.	CARD CHAR.	PUNCH	OCT.	DEC.	4020 CHAR.
0	0	60	48	0	E	12-5	25	21	E	$	11-3-8	53	43	$
1	1	01	01	1	F	12-6	26	22	F	*	11-4-8	54	44	*
2	2	02	02	2	G	12-7	27	23	G	none	none	55	45	γ
3	3	03	03	3	H	12-8	30	24	H	none	none	56	46	~
4	4	04	04	4	I	12-9	31	25	I	none	none	57	47	đ
5	5	05	05	5	none	none	32	26	π	blank	none	00	00	blank
6	6	06	06	6	.(period)	12-3-8	33	27	.(period)	/	0-1	61	49	/
7	7	07	07	7	)	12-4-8	34	28	)	S	0-2	62	50	S
8	8	10	08	8	none	none	35	29	β	T	0-3	63	51	T
9	9	11	09	9	none	none	36	30	±	U	0-4	64	52	U
none	none	12	10	∂	none	none	37	31	?	V	0-5	65	53	V
=	3-8	13	11	=	-	11	40	32	-	W	0-6	66	54	W
'	4-8	14	12	"	J	11-1	41	33	J	X	0-7	67	55	X
none	none	15	13	'	K	11-2	42	34	K	Y	0-8	70	56	Y
none	none	16	14	δ	L	11-3	43	35	L	Z	0-9	71	57	Z
none	none	17	15	α	M	11-4	44	36	M	:	0-2-8	72	58	o
+	12	20	16	+	N	11-5	45	37	N	,	0-3-8	73	59	,
A	12-1	21	17	A	Ø	11-6	46	38	Ø	(	0-4-8	74	60	(
B	12-2	22	18	B	P	11-7	47	39	P	none	none	75	61	∫
C	12-3	23	19	C	Q	11-8	50	40	Q	none	none	76	62	Σ
D	12-4	24	20	D	R	11-9	51	41	R	none	none	77	63	◻
					none	11-0	52	42	.(pl.pt.)					

See also: List in Chapter 1, page 17; Table III, Chapter 3, page 66; and Table VIII, Chapter 6, page 151

TABLE 9.2

IVY SYMBOLS

1. Special symbols: A (stored addresses), L (local references within formulas), X (index registers). See Chapter 2, pages 28-29.

2. Program defined symbols: One to six alphabetic characters <u>except</u> A, X, and L. Must be defined on "D" cards unless:

   a. Formula name
   b. Single character used for renaming index register
   c. Name of remark

   See Chapter 1, page 17 and Chapter 3, pages 47-49.

3. Numbered symbols: a symbol of type 2 followed by numeric digits. Defined on "S" cards and used only for names of remark and calling sequence blocks. See Chapter 1, page 18, Chapter 2, page 29, and Chapter 3, pages 61-67.

4. Internal IVY symbols: begin with "$" and consist of 0, 1, or 2 alphanumeric characters. These symbols are as follows:

SYMBOL	MEANING AND USE	PAGE REFERENCE
$	Sign; address modifier and operation (e.g. "+ $," etc.)	69,81
$W	Control word; address modifier	82
$WA	Control word address; address modifier	82
$WC	Control word count; address modifier	82
$WP	Control word position; address modifier	82
- - -	- - - - - - - - - - - -	- - -
$CA	Convert arithmetic; operation	69,87
$CX	Convert exponent; operation	69,87
$U	Set result; operation	69,87
$V	Set result; operation	69,87

TABLE 9.2 (Continued)

SYMBOL	MEANING AND USE	PAGE REFERENCE
\$R	Square root; operation	69,87
\$X	Set result; operation	89
\$Q	"Q" portion; expression modifier	83
\$T	"T" portion; expression modifier	83
\$D,\$DA,...\$DZ	Address; subroutine data blocks	85,110
\$CS	Address; calling sequence block	84
\$L$_n$	Address; machine information	85
\$Z	Address; zero	85
\$M	Address; quantity to left of "="	84
\$E	Execute; instruction to compiler	115
\$J	Jump; instruction to compiler	55,104
\$P	Pathfinder; special register	101,103,104
\$AP	Assembly program; subroutine	119
\$CM	Character manipulation; subroutine	154
\$DK	Disk program; subroutine	152
\$LD	Loading program; subroutine	119
\$MP	Microfilm plot; subroutine	147
\$ØP	Instructions to operator; subroutine	153
\$ØD	Octal dump; subroutine	129
\$PH	Punch; subroutine	126
\$PR	Print; subroutine	131
\$SW	Switch test; subroutine	126
\$TP	Tape program; subroutine	121
\$TT	Trigger test; subroutine	128

TABLE 9.3

CARD FORMATS

1	Format	PAGE REFERENCE
*	* * * Min(3) name,ph.(19) no.(4) code(3) group(3) cat(2) $2\overset{G}{H}$ MOOO tapes(2)	22-23
B	JØB,IØD,REEL, etc.	23-26
S	(N_1),$A(N_2)$,$X(N_3)$,$L(N_4)$,SYMBOL,(N_5),...	26
P	S	30
D	SYMBØL,SYMBØL=Q,SYMBØL($N_1,N_2,\ldots,N_n$),SYMBØL($N_1,\ldots,N_n$)=$Q_1,\ldots,Q_m$	43-57
R	NAME(P) = HØLLERITH^CHARACTERS^$$$	61-64
K	NAME(P) = ($CSW_1:CSW_2:\ldots:CSW_n$)	64-67
Ø	INSTRUCTIONS	33
T	$TSCW_1:TCSW_2:\ldots:TCSW_n$	33-35
A	$RDH $WRN,F	35-37
I	Algebraic code, etc.	68-117
L	Longhand code, etc.	Appendices 2,3
F	AD:$B,L:$A,M	38-39
X	NAME ØF FØRMULA SET OR BLANK	39-40
E	SYMBØL = $Q_1,Q_2,\ldots,Q_m$,SYMBØL(Q.P) = $Q_1,Q_2,\ldots Q_i$, SYMBØL(T.P) = $Q_1,\ldots,Q_i$	57-59

See also Table I, page 42, and Table II, pages 60-61.

TABLE 9.4

ALGEBRAIC OPERATIONS

FLOATING POINT	FIXED POINT	INDEX REGISTER
+	+	+
-	-	-
*	*	.$X
**	**	
/	/	
//	//	
+ $	+ $	Boolean
- $	- $	+
* $	* $	*
.$R	.$R	'
.$CA	.$CA	.$U
.$CX	.$CX	.$V
.$U	.$U	
.$V	.$V	

See also: Floating point, page 69
Fixed point, page 86
Index register, page 89
Boolean, page 93

RULE: All operations are considered from left to right, i.e., each operation takes in all expressions so far computed on the left to the right of a left parenthesis or equal sign, whichever occurs latest.

TABLE 9.5

SYMBOLS, EXCLUSIVE OF MODIFIERS, ALLOWED IN ALGEBRA

A. Symbols allowed on either side of equal sign

TYPE OF SYMBOL	MEANING	NOTE
SYMBØL	Name of data block or first element	1
SYMBØL_n	nth element of data block	1
NAME	Name of "K" block--numbered or not	2
\$CS, $\$\text{CS}_n$	Calling sequence block	3
\$D,\$DA,...\$DZ	Subroutine data blocks	3
$\$\text{D}_n, \$\text{DA}_n, \ldots, \$\text{DZ}_n$	nth element of subroutine data blocks	3
\$L3	NLA (next loading address)	4
\$L4	NBA (next block address)	4
\$Z	Zero address	5
Xn	Index register n	6

B. Symbols allowed only on right side of equal sign

\$L1	FAC (first address for code)	7
\$L2	FAD (first address for data)	7
\$L5	Machine number	7
\$L6	No. of BCD characters per word	7
\$M	Quantity on left of equal sign	6

NOTES: (Types of symbol modifiers allowed - see Table 9.6):

1. Either side of equal sign: type A; or type E (except "\$WP")
   left of equal sign: type A and type B
   right of equal sign: "\$WP" or type A and one of types C,D, or F.

2. Same types of modifiers as in note 1 with the addition that type A can occur with "\$WA" and "\$WC" on left or right of equal sign.

3. Same types of modifiers as in note 1 except type E is not allowed.

4. Either side of equal sign: type A
   left of equal sign: type A and type B
   right of equal sign: type A and type F.

TABLE 9.5 (Continued)

NOTES:

5. Modifiers required.

   Either side of equal sign: type A or type A plus "$WA" or "$WC."
   left of equal sign: type A and type B
   right of equal sign: type A and one of types C, D or F

6. No modifiers allowed.

7. Type A or type A and one of types C, D or F.

TABLE 9.6

ADDRESS (OR SYMBOL) MODIFIERS M: "SYMBØL(M)"

MODIFIER	TYPE	MEANING	LEFT OF=	RIGHT OF=	BOOLEAN	FIXED PT.	FL.PT.
P	A	Pth element	yes	yes	yes	yes	yes
X_n+P	A	above plus index	yes	yes	yes	yes	yes
X_n	A	index register	yes	yes	yes	yes	yes
A_n	A	stored address	yes	yes	yes	yes	yes
X_n+A_n	A	above plus index	yes	yes	yes	yes	yes
A	B	fixed pt.expression	yes	no	no	yes	no
B	B	Boolean expression	yes	no	yes	no	no
Q.P	C	"Q" of double-stored no.	no	yes	yes	yes	yes
M.P.	C	magnitude of above	no	yes	yes	yes	yes
T.P.	C	"T" of Double-stored no.	no	yes	yes	yes	no
M	D	magnitude of no.	no	yes	yes	yes	yes
$	D	sign only of no.	no	yes	no	yes	yes
R	D	save low order part	no	yes	no	yes	yes
$W	E	control word	yes	yes	yes	yes	no
$WA	E	control word address	yes	yes	yes	yes	no
$WC	E	control word count	yes	yes	yes	yes	no
$WP	E	control word position	no	yes	yes	yes	no
S	F	swap	no	yes	yes	yes	yes

See also pages 74-83

TABLE 9.7

EXPRESSION MODIFIERS (RIGHT OF EQUAL SIGN ONLY)

MODIFIER	MEANING
.ØQ.P	Put result in "Q" part
.ØT.P	Put result in "T" part; may follow only fixed point expressions

See also pages 83-84.

TABLE 9.8

REFERENCE POINT ENTRIES AND BRANCHES

A. Reference Point Entries

ENTRY	MEANING
X_n(1,P)	Forward index loop entry
X_n(P,1)	Backward index loop entry
L_n	Local L-entry for references within formula
L_n.X_n	Local subroutine. Contents of "$P" go to X_n.
FØRM	Formula entry for references within formula set. "FØRM" not on "D" card.
FØRM.X_n	Formula subroutine C($P) $\rightarrow X_n$
FS	Formula set entry for references from entire code. "FS" on "D" card
FS.X_n	Formula set subroutine.C(SP) $\rightarrow X_n$

B. Types of Branches

(X_n)	Loop on index X_n to nearest previous loop entry for index X_n
(L_n) (FØRM) (FS)	Unconditional branches to L-entry in same formula, formula in same set, and formula set, respectively.
(L_n)Algebra=C (FØRM)Algebra=C (FS)Algebra=C	Conditional branches to L-entry in same formula, formula in same set, and formula set, respectively
($P,$L_n$) ($P,FØRM) ($P,FS)	Pathfinder branches to local subroutine in same formula, formula subroutine in same set, and formula set subroutine, respectively
(X_m + N)	Return to instruction after calling sequence of subroutine

TABLE 9.9

CALLING SEQUENCE CONVENTIONS

(Each entry represents a legal calling sequence word entered between colons on "K" card or after pathfinder branch)

ENTRY	MEANING
¢XXX	1,2, or 3 alphanumeric characters in "¢WC" portion. Used for control purposes.
P	Parameter algebra expression. Result is computed and placed in "¢WA" portion or more, if greater than 2^{15}.
AD(¢W)+P	"AD" any programmer-defined symbol. CW(AD) modified by P is placed in entire word.
AD(¢WA)+P	Control word address modified by P is placed in "¢WA" portion.
AD(¢WC)+P	Control word count modified by P is placed in "¢WA" portion.
AD(¢WP)	Control word position is placed in "¢WA" portion.
AD(P)	Contents of AD(¢WA)+P are placed in "¢WA" portion.

In addition, the following compound entries are allowed:

¢XXX,P (P < 2^{18})
¢XXX,AD(¢WA)+P
¢XXX,AD(¢WC)+P
¢XXX,AD(¢WP)
¢XXX,AD(P) (C(AD(¢WA)+P) fixed pt. $\leq 2^{18}$)

TABLE 9.10

SUMMARY OF CALLING SEQUENCES TO IVY SUBROUTINES

A. ∅AP

Calling seq. word	MEANING
∅RDN,F	Read in and convert code in file F, tape N

B. ∅TP

Calling seq. word(s)	
∅HDX	Set tape "X" to high density
∅LDX	Set tape "X" to low density
∅RWX	Rewind tape "X" to load point
∅ULX	Rewind tape "X," then unload
∅EFX	Write end-of-file on tape "X"
∅ETX	Write end-of-tape record on tape "X"
∅BBX,P	Backspace tape "X" through P records
∅BFX,P	Backspace tape "X" through P files
∅FBX,P	Forward space tape "X" through P records
∅FFX,P	Forward space tape "X" through P files
∅RDX,AD(∅WA)+P	May be last calling sequence word only. If ID of current record on tape "X" = contents of location specified, ∅CS1 = 1. If not, ∅CS1 = 0.
∅RDX,AD(∅WA)+P: AE(∅WP)	Reads into block "AE" the record with ID = C(AD(∅WA)+P), from tape "X"
∅P∅	Sets "∅TP" to run in parallel mode
∅S∅	Sets "∅TP" to run in serial mode

For further reference see Table V , page 120(tape numbers), and Chapter 6, pages 119-126.

C. ∅TT

Calling sequence words	
∅P, AD(∅WP):∅R	Print a diagnostic comment if any indicators are on, and return control to problem program

TABLE 9.10 (Continued)

C. $\not{\$}$TT (continued)

Calling sequence words	MEANING
ØN:ØR	No print; return to problem program
ØP.AD(ØWP):ØI	Print comment and return to IVY if any indicators on
ØN:ØI	No print; return to IVY if any indicators on

D. ØØD

Calling sequence word	
ØDA	Dump all data and longhand code
SYMBØL(SWP)	Dump data block or longhand formula set named

E. ØPH,ØPR

Calling sequence word(s)	SPR MEANING	SPH MEANING
Ø1P	Restore page (on-line)	Insert blank card
Ø2P	Half-page skip (on line)	Insert blank card
ØDP	Double space (on line)	Insert blank card
Ø1T	Restore page (off-line)	Insert blank card
Ø2T	Half-page skip (off-line)	Insert blank card
ØDT	Double space (off-line)	Insert blank card
Ø1M	Restore page (microfilm)	Insert blank card
Ø2M	Half-page skip (microfilm)	Insert blank card
ØDM	Double space (microfilm)	Insert blank card
ØP,REM(ØWP)	Print remark on-line	Punch rem. on-line
ØT,REM(ØWP)	Print remark off-line	Punch rem. off-line
ØM,REM(ØWP)	Print remark on microfilm	Punch rem. off-line
ØF,FORMAT(ØWP)	Format statement (see note below)	Same
REMARK(ØWP)	Print remark as specified by previous format	Punch same
VECTOR(ØWP)	Print vector specified by previous format	Punch same, or matrix or array
ØPN,PARAM(ØWA)+P	Print N parameters as specified by previous format	Ignored
Ø2,MATRIX(ØWP):P	Print matrix (with 1st dimension P) as specified by previous format	Punch same (Ø2,P are superfluous)

TABLE 9.10 (Continued)

E. ∮PH,∮PR (continued)

Calling sequence word(s)	SPR MEANING	SPH MEANING
∮A,ARRAY(SWP):P_1:P_2	Print array(1st 2 dimensions P_1,P_2) as specified by previous format)	Punch same (∮A,P_1, P_2 are superfluous)
VECTØR(∮WP):∮D,P	Print both Q and T portions	Punch same, or matrix or array
VECTØR(∮WP):∮Q_1,P	Print Q portion only of DS vector, tag length P	Punch same, or matrix or array
VECTØR(∮WP):∮T,P	Print T portion only of DS vector, tag length P	Punch same, or matrix or array
∮2,MATRIX(∮WP):∮D,P_1:P_2	Print both Q and T portions of DS matrix, tag length P_1, 1st dimension P_2	Punch same (∮2,P_2 are superfluous
∮2,MATRIX(∮WP):∮Q,P_1:P_2	Print Q portion only of DS matrix	Punch same (∮2,P_2 are superfluous)
∮2,MATRIX(∮WP):∮T,P_1:P_2	Print T portion only of DS matrix	Punch same (∮2,P_2 are superfluous)
∮A,ARRAY(∮WP):∮D,P_1: P_2:P_3	Print both Q and T portions of DS array,tag length P_1, 1st 2 dimensions P_2,P_3	Punch same (∮A,P_2,P_3 are superfluous)
∮A,ARRAY(∮WP):∮Q,P_1: P_2:P_3	Print Q portion only of DS array	Punch same (∮A,P_2,P_3 are superfluous)
∮A,ARRAY(∮WP):∮T,P_1: P_2:P_3	Print T portion only of DS array	Punch same (∮A,P_2,P_3 are superfluous)

Note on format statements: the general appearance of a format statement is as follows:

R|FØRMAT = $C_1C_2\cdots C_n$,(IMM.REM.$_1$)$N_1 \cdot S_1 \cdot I_1 \cdot F_1 \cdot E_1$,...,(IMM.REM$_m$)$N_m \cdot S_m \cdot I_m \cdot F_m \cdot E_m$,∮∮∮

where

each "C_i" represents one of the following conditions:

TABLE 9.10 (Continued)

C_i	SPR MEANING	SPU MEANING
P	Print on-line	Punch on-line
T	Print off-line	Punch off-line
M	Print on microfilm	Punch off-line
C	Print column indices	Ignored
R	Print row indices	Ignored
L	Print in line format	Ignored
F	Print fixed point numbers as integers	Punch fixed point nos. as integers
∅	Print tags of DS nos. in octal	Punch tags of DS nos. in octal

The "IMM.REM$_i$" are remarks consisting of up to 119 hollerith characters, with the single restriction that parentheses must occur in closed pairs. These remarks are printed preceding the numbers governed by the format, and are ignored by the punch program.

Each "N.S.I.F.E." field controls the printing or punching of one or more blocks of numbers as follows:

NUMBER	
N	Number of blocks controlled by this field.
S	Number of blank spaces preceding each number (ignored by "$PU"). must be $\leq$ 119.
I	Number of integer digits (i.e., digits to the left of the decimal point). must be $\leq$ 15.
F	Number of fraction digits (i.e., digits to the right of the decimal point). must be $\leq$ 15.
E	Number of digits in exponent. must be $\leq$ 15.

For further details, see pages 131-147.

TABLE 9.10 (Continued)

F. $MP

Calling sequence word(s)	MEANING
$AFN,P	Advance film; if N = 1, set for hard copy; advance "P" more frames.
$SG:$P_1$:$P_2$	Select grid, horizontal option "P_1," vertical option "P_2."
$LG:XYBDS($WA)+P	Valid only after above entry. Label grid, assuming X_{min},X_{max},Y_{min},Y_{max} stored in that order starting at "XYBDS($WA)+P."
$WRN,REM($WP):R($WA)+$P_1$: C($WA)+P_2	Write remark "REM" horizontally (N=0) or vertically (N=1) starting at row and column position specified.
$PFN,M:XC($WP):YC($WP): XYBDS($WA)+P:TEMP ($WP)	Plot XC versus YC, using char. M, and connect with line if N=1. XYBDS as above; "TEMP" = block same length as XC,YC for erasable.
$GA,XYBDS($WA)+P:XZERØ($WA)+P: YZERØ($WA)+P	Generate a pair of axes through XZERØ, YZERØ;XYBDS as above.

See also pages 147-152.

G. $DK

Calling sequence words	
$WRN,ID($WA)+P:DATA($WP)	Write "DATA" block on disk with ID as specified. N = tape number on machines with no disk, 0 otherwise.
$RDN,ID($WA)+P:DATA($WP)	Read block with ID specified into "DATA." N as above .

H."$ØP" has one calling sequence word, of the form

REMARK($WP)

The remark specified is printed on-line. See page 153.

TABLE 9.10 (Continued)

I. ¢CM

Calling sequence word(s)	MEANING
¢SN,REMARK(¢W)	Set remark specified to the character represented by decimal number "N" (see Table 9.1).
¢MN:AD(¢WA)+P_1:REM1(¢WP): AE(SWA)+P_2:REM2(¢WP)	Move N characters from REM1 to REM2, positions as specified.
¢CN:AD(¢WA)+P_1:REM1(¢WP): AE(SWA)+P_2:REM2(¢WP)	Compare N characters of REM1 and REM2 positions as specified; if equal ¢CS1=1, otherwise ¢CS1=0.

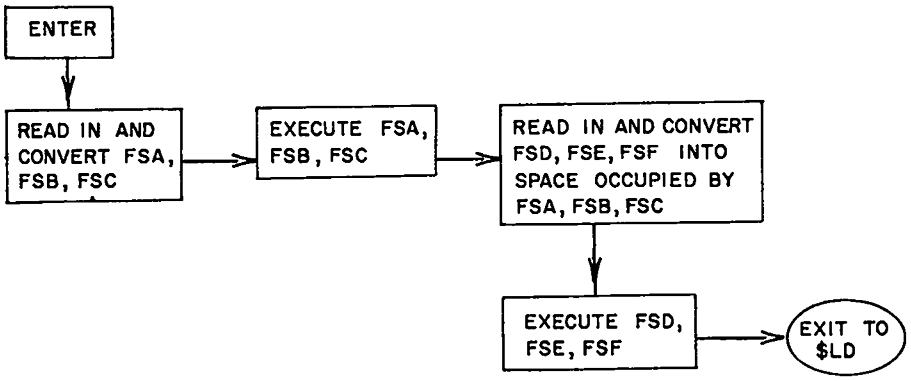
See pages 154-156.

APPENDIX 1

MANIPULATING THE SYMBOL TABLE

The main function of this Appendix is to give a detailed description of how the symbol table can be altered and used in computations involving only the IVY algebraic instruction set without recourse to longhand instructions. More sophisticated computations can be performed using longhand instructions, but in general these will not be necessary. Presumably symbol table manipulations will seldom if ever be used by most programmers; they are useful only when codes are very long and complicated and extreme methods of conserving core storage are necessary.

Handling a program which is too large to fit in the machine. It often happens that the whole of an unusually long program, or one with a large amount of data, will not fit into core at one time. A code of this size, however, is quite easy to handle in the IVY system. First, the code must be organized in a specific fashion: there must be a flow code, in this case usually called a "master code," which is in core at all times, and which controls access to and conversion of the remainder of the code, only a fraction of which is in core at one time. In this case we assume that control passes through each of several portions of the code only once. As an example of this, we shall consider a program consisting of a master code and six formula sets "FSA," "FSB,",...,"FSF," to be executed in this order of which only three can occupy core at one time. The master code for handling such a case can be diagrammed as follows:


ENTER
READ IN AND CONVERT FSA, FSB, FSC
EXECUTE FSA, FSB, FSC
READ IN AND CONVERT FSD, FSE, FSF INTO SPACE OCCUPIED BY FSA, FSB, FSC
EXECUTE FSD, FSE, FSF
EXIT TO $LD

If we assume that the six formula sets occupy files 2-7 on tape 2, the code for doing this looks approximately as follows:

Line No.			NOTES
1	I	MASTER,($P, $AP: $RD2,2: $RD2,3: $RD2,4),	1
		⋮	
2	C	CØDE ^ EXECUTES ^ FSA,^ FSB,^ FSC	2
		⋮	
3	I	$L3 = FSA($WA)+1,($P, $AP: $RD2,5: $RD2,6: $RD2,7),	3
		⋮	
4		CØDE ^ ØMITTED ^ EXECUTES ^ FSD,^ FSE,^ FSF	4
		⋮	
5	I	($P, $LD),	5

NOTES:

1. The master code, appropriately named "MASTER," enters "$AP to read in and convert "FSA," "FSB," and "FSC."

2. "FSA," "FSB," and "FSC" are executed, the flow of control being by means of conditional "L" branches and pathfinder branches peculiar to the program.

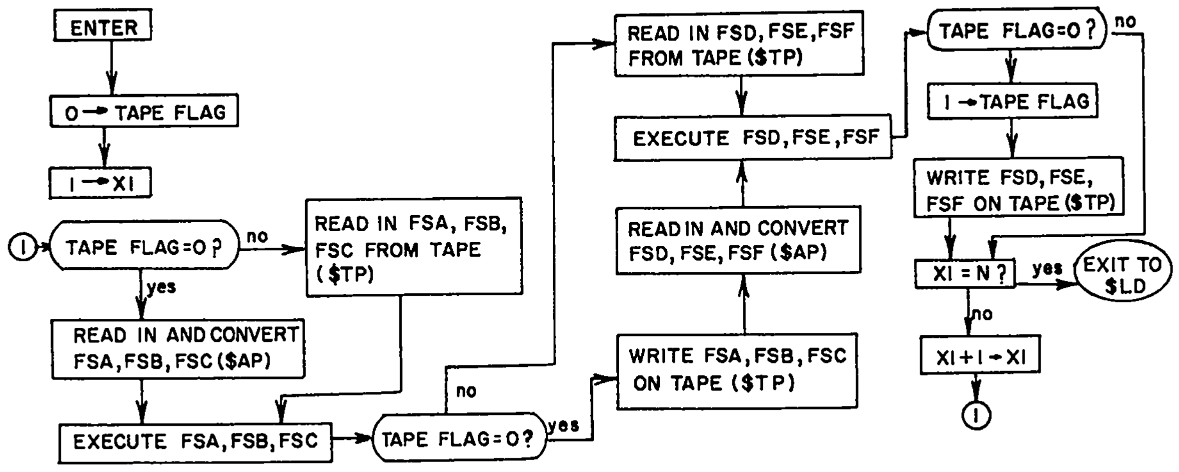
3. Next, "$L3," which contains the next loading address (NLA) for code, is reset to the value it had before "FSA" was loaded, and the three formula sets "FSD," "FSE," and "FSF" are read in and converted, covering the previous three formula sets.

4. The second section of the code is executed.

5. At the end of the program, a pathfinder branch to "$LD" is executed.

In addition to the master code, of course, there may be other formula sets which are always in core, such as subroutines referred to by both sections of the code. Also, this simple example can easily be extended to the case where the code must be divided into more than two sections. It is necessary merely to repeat a similar sequence of instructions to that on line 3 as many times as needed.

How to "ping-pong" a code. By "ping-ponging" a code it is meant that several sections of the code must be repeatedly read in and out of core since each must be executed more than once, unlike the previous example. Thus the various sections of the code bounce in and out of core like a ping-pong ball. To do this, after each section of the code is converted and executed, it must be written on tape using "$TP;" then it need only be read in from tape for each succeeding execution. This method saves considerable compiling time since on successive executions the code need not be re-compiled. If we consider again a case in which a code contains six formula sets, of which only three can fit into core, and assume each section is to be executed N times, the following diagram illustrates how the master code should look.


ENTER
0 → TAPE FLAG
1 → XI
1
TAPE FLAG = 0 ?
no
yes
READ IN FSA, FSB, FSC FROM TAPE ($TP)
READ IN AND CONVERT FSA, FSB, FSC ($AP)
EXECUTE FSA, FSB, FSC
TAPE FLAG = 0 ?
no
yes
WRITE FSA, FSB, FSC ON TAPE ($TP)
READ IN AND CONVERT FSD, FSE, FSF ($AP)
READ IN FSD, FSE, FSF FROM TAPE ($TP)
EXECUTE FSD, FSE, FSF
TAPE FLAG = 0 ?
no
1 → TAPE FLAG
WRITE FSD, FSE, FSF ON TAPE ($TP)
XI = N ?
yes
EXIT TO $LD
no
XI + 1 → XI
1

We note that a "tape flag" is necessary, so that the master code can detect whether or not the two sections of the code have been written on tape.

IVY DATE PAGE NAME PROBLEM

73	74	75	76	77	78	79	80

1 2 72 CODE

```
I MASTER, TF=0, XI(1,N), (L1)TF=NZ,
  ($P, $AP: $RD2,2:$RD2,3:$RD2,4), (L2),
  L1, ($P, $TP: $RW3: $RD3, T($WA)+1: FSA($WP):
  $RD3, T($WA)+2: FSB($WP): $RD3, T($WA)+3:FSC($WP)),
  L2,...
C ØMITTED ^CØDE ^ EXECUTES ^ FSA,^ FSB,^FSC
  :
I (L3)TF=NZ, ($P, $TP: $RW3: $WR3, T($WA)+1: FSA($WP):
  $WR3, T($WA)+2: FSB($WP): $WR3, T($WA)+3: FSC($WP)),
  $L3 = FSA($WA)+1, ($P, $AP: $RD2,5: $RD2,6: $RD2,7),
  (L4), L3, ($P, $TP: $RD3, T($WA)+4: FSD($WP):
  $RD3, T($WA)+5: FSE($WP): $RD3, T($WA)+6: FSF($WP)),
  L4,...
C ØMITTED ^ CØDE ^ EXECUTES ^ FSD,^ FSE,^ FSF,
  :
I (L5)TF=NZ, TF=1, ($P, $TP, $WR3, T($WA)+4:
  FSD($WP): $WR3, T($WA)+5: FSE($WP): $WR3, T($WA)+6:
  FSF($WP)),
  L5,(XI), ($P, $LD),                                   exit
```

Figure A1

The code for doing this is illustrated in Figure A1. Again it is assumed that tape 2 contains the six formula sets in unconverted form in files 2-7, and that tape 3 is a blank to be used in the "ping-ponging" operation. "T" is a block containing the ID words required by the tape program. By observing the arrows on the outside of the coding sheet, one can follow the path of flow as it corresponds to the above diagram.

The methods illustrated in this appendix thus far can be generalized, extended, or combined to handle virtually any lengthy code that can be devised. Similar methods can also be used to write blocks of converted code on the disk unit of machines which contain such a device. The speed of a disk unit is always greater than the speed of tapes, so this procedure is recommended when possible. The programmer should note that formula sets are the only units of code which can be written on tape or disk since only formula set names are contained in the symbol table. Neither <u>individual</u> formulas nor <u>groups</u> of formula sets under the control of only two calling sequence words can be written: the former, because formula names are not contained in the symbol table; and the latter, because two calling sequence word entries are necessary for <u>each</u> formula set read or written. Whereas one can write several formula sets onto the assembly tape under control of an "A" card, this is not possible using "$TP" or "$DK."

Other manipulations of the symbol table are also possible using various techniques involving the "$WC" and "$WA" modifiers, in addition to the special locations "$L1," "$L2," "$L3," and "$L4" (see Chapter 4, page 85). In general, however, manipulations which change the contents of the symbol table are to be avoided if at all possible since they might cause unpredictable difficulties in IVY. Therefore no examples of this type of programming are given.

APPENDIX 2

THE 7090 LONGHAND INSTRUCTION SET

The 7090 longhand instruction set is divided into classes according to the types of addresses and decrements (if any) allowed. The reader is referred to the IBM Reference Manual, 7090 Data Processing System, for descriptions of the operations and coding examples. First in this appendix we shall discuss general addressing conventions, address modifiers, and other conventions allowed in longhand coding, then the list of allowed instructions by classes, and finally a coding example will be given.

Addressing conventions. In the description of each class of operations, the symbol "AD" is used to represent the allowed address symbols for the particular class. Each class may use a subset of the following set of address symbols:

SYMBØL	MEANING
NAME	The symbolic name of a data, remark, or calling sequence block. This may be the name of a numbered block.
NAME_n	The nth element of data block "NAME."
∅CS	The calling sequence block.

SYMBØL	MEANING
$\not{C}S_n$	The nth element of the calling sequence block.
$\not{D}$	Any of the subroutine data blocks.
$\not{D}_n$	The nth element of any of the subroutine data blocks.
A_n	Address of instruction with "A_n" modifier.
$\not{L}_n$	One of the elements of the "$\not{L}$" block.
$\not{Z}$	Zero or null address.
L_n	Name of an "L" entry.
F	Formula name.
FS	Formula set name.
$\not{S}R$	Name of an IVY subroutine.
*	Location of current instruction.
X	Fixed or floating point literal (Decimal only).
N	Absolute address equal to the number N.

In each class of operations, address modifiers (which occur within parentheses following the address symbol) are represented by "M_i," and each symbol is allowed a subset of the following set of modifiers:

M_i	MEANING
M_o	No modifier
M_1	P (parameter algebra)

$\underline{M_i}$	MEANING
M_2	X_n + P (parameter algebra plus index register)
M_3	X_n (index register)
M_4	A_n (stored address)
M_5	X_n + A_n (index register plus stored address)
M_6	ØWP (control word position)
M_7	ØWC/ØWA (control word count or address)

Longhand instructions are separated from one another by colons. The translation to machine language is strictly one-to-one, i.e., each entry between colons corresponds to one machine word, except for certain entries which represent instructions to the <u>compiler</u>, which will be noted as they occur. The elements of an instruction are separated from one another by commas and sets of closed parentheses. The general format of a longhand instruction is:

$$:ØP(I/X_n),AD(M_i),DCR/CT:$$

Conventions are explained in the discussion of each class.

<u>General form of a longhand code</u>. Longhand instructions are entered on "L" cards, which may be intermixed with discretion in any desired manner with "I" cards, to supplement the algebraic code in as great or small a manner as desired. However, in this appendix we shall consider longhand formula sets, i.e., <u>complete</u> routines or subroutines coded in

longhand. The programmer interested only in inserting a card or two of longhand code into his program should have no difficulty once the larger concepts have been studied. See restriction 2, page 217.

Longhand codes are divided into formula sets, formulas, and local entries just as are algebraic codes. The same rules for referencing discussed in Chapter 5 also apply to longhand codes. However, the methods of forming entry points and branches of course differ between the two systems. Formula set names, formula names, and L-entries precede the operation at the entry point in parentheses, thus:

(FS) ∅P, AD(M_i)

(F) ∅P, AD(M_i)

(L_n) ∅P,AD(M_i)

In branches to these entries, the name of the entry occurs in the address of the instruction outside parentheses, modified if necessary, e.g.,

TRA,FS(M_i)

TRA,F (M_i)

TRA,L_n(M_i)

and so on for other types of transfer instructions.

<u>Subroutine conventions</u>. Internal IVY subroutines or subroutines coded in the algebraic language must be entered by the instruction

TSX(∅P), NAME

where "∅P" is the pathfinder register and "NAME" is the name of the subroutine. On the 7090 the pathfinder register "$\not{\text{\$}}$P" is synonymous with index register 3, X3. Thus the following entry is equivalent to the above:

```
TSX(X3), NAME
```

If a subroutine written in longhand is entered by a pathfinder branch from a portion of the code in algebraic language, or by means of one of the above instructions, one can use the first instruction of the subroutine to store the pathfinder contents in the address or decrement of a specified location "AD" as follows:

```
(NAME) SXA(¢P),AD   or (NAME) SXD(¢P),AD
```

which, of course, are equivalent to the entries

```
(NAME) SXA(X3),AD  or  (NAME) SXD(X3),AD
```

Calling sequences. Calling sequences may be constructed after a "TSX" to any subroutine by using the pseudo-operation "CSW" and by following it with a standard IVY calling sequence word entry, separating the calling sequence words from each other by colons. For example:

```
L|TSX(¢P),NAME:CSW,¢ABC,GE(¢WP):CSW,¢CT2,CRA(¢WC):CSW,3+7*GE: ...
```

The IVY subroutines with variable length calling sequences (namely ¢AP, ¢TP,¢ØD,¢PR,¢PH,¢MP,¢DK, and ¢CM) require an "HTR" with null address field following the last calling sequence word. This is to signal the subroutine that the calling sequence has ended. The compiler automatically inserts this zero word in calling sequences to these routines written in the algebraic code, but this is not done automatically in the longhand system because, as mentioned previously, longhand code is translated in a one-to-one fashion. Thus, for instance, to print a vector, the following statements should be used:

```
L|TSX($P),$PR:CSW,$F,FØRMAT($WP):CSW,VECTØR($WP):HTR:...
```

Definition of "$D" blocks. To define "$D" blocks at the beginning of a formula set coded in longhand, one must enter these definitions between a single set of colons immediately following the first instruction of the formula set, for example:

```
L|(FS)SXD($P),AD:$D(4),$DC(3) = 2.156,3.172,5.171,
 |$DH(5,9,10),$DM(50) = 5.152,(S)47,5.213,5.215:...
```

The format for defining "$D" blocks is otherwise the same as the format for defining them on "I" cards, discussed in Chapter 5, pages 110-112. Since "$D" definitions represent instructions to the compiler, a word of code is not, in this case, assembled to correspond to the definitions between the pair of colons.

Renaming. One may rename an index register with an alphabetic symbol in the same manner as on "I" cards, by entering between colons as follows:

$$:\!*\text{SYMBØL} = X_n:$$

Again, since this entry represents an instruction to the compiler, no information is assembled.

Storing addresses. The "A_n" symbols may be used in the address fields of "store address" instructions, and the address being stored will be placed in instructions having addresses modified by this same "A_n" symbol. In longhand coding, instead of the entry "$A_{n.m}$" specifying m store address instructions, one must instead write down consecutively, all m instructions. The number of store address instructions entered must equal the number of instructions in which the address is stored. For example:

```
L|CLA,FRN(ØW):ADD,FRNX(X1):STA,A1:STA,A1:STA,A1:...
 |...LDQ,FRN(X2+A1):FMP,FRN(X3+A1):...:STØ,FRN(X2+A1):...
```

Here three "STA" instructions with address "A1" are present, and three instructions with addresses modified by "A1" are also present. Note that instructions in which addresses are stored must occur sequentially after the "STA" instructions referring to them. The "STA" instructions must occur sequentially together, as in the above example.

In longhand coding, one can also store addresses in instructions at or near local, formula, or formula set entries, or near the "STA" itself, by using the name of the entry or "*" with appropriate modifiers. "A_n" is not needed in such a case. For example:

```
L|(L1)CLA,FRN(X1):SUB,FRN(X2):STØ,FRN(X3):...
 |...CLA,FRN(ØW):ADD,FRNX(X1):STA,L1:STA,L1(1):STA,L1(2):...
```

This technique can thus be used to store addresses in a backward direction, if desired.

Storing decrements, prefixes, tags, and left half of MQ. If these items are stored in code, then "A_n" symbols, which are reserved for "STA" alone, cannot be used. Instead the addresses of "STD," "STP," "STT," and "SLQ" instructions must contain the name of a local, formula, or formula set entry, or the "*," with appropriate modifiers, in the same manner as "STA" instructions which do not use "A_n" symbols.

Further restrictions on longhand coding. The following check list summarizes the remaining restrictions on longhand coding:

1. To repeat what was said previously, only three index registers, X1, X2, and X3, are allowed in 7090 longhand coding.

2. The programmer should note that X3 is used by the algebraic code not only to simulate the pathfinder, but also to simulate index registers X4, X5, etc. If the longhand code consists only of a few cards inserted in the middle of an algebraic code, the programmer must save the contents of X3 or of the last index register above X3 to which he has referred, to ensure that its previous contents, if needed after the longhand segment, are not destroyed. This can be done in two ways:

   a. In the algebraic code:

      I|....,T1 = X_n,

      L|....(Longhand code)...

      I|X_n = T1,...

      where "X_n" is the last index register above X3 referred to in the algebraic code and "T1" is some suitably chosen temporary location; or

   b. In the longhand code:

      I|....(Algebraic code)

      L|SXA(X3),T1:...(Longhand code)...

      L|....:LXA(X3),T1:...

      I|....(Algebraic code)

      The previous contents of X3, simulating some higher index register, are preserved by the longhand code.

In general, short longhand code inserts in algebraic code should be used with great discretion. Because of the simulation of extra index registers by X3, one must, as remarked above, take great care not to destroy the contents of this index register. Branching between such "pockets" of longhand code and the algebraic code should also be avoided. The unwary coder should in general follow these rules:

a. Short longhand inserts in a predominantly algebraic code should not use X3 and should not be entered or left by branches to algebraic code, without the use of great care.

b. Preferably, the smallest unit of longhand code in an algebraic code should be at least a formula.

3. Only one index register can be specified in an instruction, i.e., the "oring" feature of the 7090 is not permitted. This is because of the possibility of wiring the 7090, at some future date, to contain seven index registers in which case this feature would no longer be present.

4. An IVY 7090 longhand instruction, together with its address, modifiers, etc., must be complete on one card. For all practical purposes, the end of the card is treated as a colon. Thus, the last instruction on a card need not be followed by a colon.

<u>Class 1.</u> <u>Arithmetic operations</u>

General format: ØP(I),AD(M_i)

The "I" is entered if indirect addressing is desired.

$\mathrm{AD}(M_i) = \mathrm{NAME}(M_0M_1M_2M_3M_4M_5M_6)^{*}, \mathrm{NAME}_n(M_0M_1M_2M_3), \not{S}\mathrm{CS}(M_0M_1M_2M_3),$

$\not{S}\mathrm{CS}_n(M_0M_1M_2M_3), \not{S}\mathrm{D}(M_0M_1M_2M_3), \not{S}\mathrm{D}_n(M_0M_1M_2M_3), \not{S}\mathrm{L}_n(M_0),$

$\not{S}\mathrm{Z}(M_0M_1M_2M_3M_4M_5), \mathrm{L}_n(M_0M_1M_2M_3M_5), \mathrm{F}(M_0M_1M_2M_3M_5),$

$\mathrm{FS}(M_0M_1M_2M_3M_5M_6), *(M_0M_1M_2M_3), \mathrm{X}, \mathrm{N}(M_3).$

---

[*] This notation means that any <u>one</u> of the modifiers shown is permitted.

OPERATIONS IN CLASS 1

MNEMONIC	MEANING	"X" IF LITERAL ALLOWED
ACL	Add and carry logically	X
ADD	Add (fixed)	X
ADM	Add magnitude (fixed)	X
ANA	"And" to accumulator (Boolean)	X
ANS	"And" to storage (Boolean)	
CAL	Clear and add logically	X
CAS	Compare accumulator to storage	X
CLA	Clear and add	X
CLS	Clear and subtract	X
DVH	Divide or halt (fixed)	X
DVP	Divide and proceed (fixed)	X
ERA	Exclusive "or" to accumulator (Boolean)	X
FAD	Floating add	X
FAM	Floating add magnitude	X
FDH	Floating divide or halt	X
FDP	Floating divide and proceed	X
FMP	Floating multiply	X
FSB	Floating subtract	X
FSM	Floating subtract magnitude	X
IIS	Invert indicators from storage	X
LAS	Logical compare accumulator and storage	X
LCHA	Load channel A	
LCHB	Load channel B	
LDI	Load indicators	X
LDQ	Load MQ	X
MPR	Multiply and round (fixed)	X
MPY	Multiply (fixed)	X
MZE	Minus zero (prefix only)	X
NZT	Storage not-zero test	
ØFT	Off test for indicators	X
ØNT	On test for indicators	X
ØRA	"Or" to accumulator (Boolean)	X
ØRS	"Or" to storage (Boolean)	
ØSI	"Or" storage to indicators	X
PZE	Plus zero (prefix only)	X
RCHA	Reset and load channel A	
RCHB	Reset and load channel B	
RIS	Reset indicators from storage	X
SBM	Subtract magnitude (fixed)	X
SCHA	Store channel A	X

MNEMONIC	MEANING	"X" IF LITERAL ALLOWED
SCHA	Store channel A	
SCHB	Store channel B	
SLQ	Store left half of MQ	
SLW	Store logical word	
STD	Store decrement	
STI	Store indicators	
STL	Store location counter	
STØ	Store	
STP	Store prefix	
STQ	Store Mq	
STR	Store location and trap	
STT	Store tag	
STZ	Store zero	
SUB	Subtract (fixed)	X
UAM	Unnormalized add magnitude	X
UFA	Unnormalized floating add	X
UFM	Unnormalized floating multiply	X
UFS	Unnormalized floating subtract	X
USM	Unnormalized subtract magnitude	X
ZET	Zero storage test	

EXAMPLES:

EXAMPLE	MEANING
STT,*(4)	Store tag in fourth instruction following
ZET,$CS1	Test contents of "SCS1" for zero
CLA,-3.25613+2	Load indicated literal into accumulator
LDQ,4(X3)	Load C(4 - C[X3]) into MQ

Class 2. Shift and sense operations

General format: ØP,AD(M_i)

where AD(M_i) = $Z($M_0M_1M_2M_3M_4M_5$),N($M_0M_3$)

OPERATIONS IN CLASS 2

MNEMONIC	MEANING
ALS	Accumulator left shift
ARS	Accumulator right shift
LGL	Long logical left shift
LGR	Long logical right shift
LLS	Long left shift
LRS	Long right shift
MSE	Minus sense
PSE	Plus sense
RQL	Rotate MQ left
SPR	Sense printer, channel A
SPT	Sense printer test, channel A
SPU	Sense punch, channel A

EXAMPLES:

EXAMPLE	MEANING
ALS,20	Shift accumulator left 20
ALS,(20)	Same as above. Note alternative form
LGL,30(X2)	Logical left 30, modified by C(X2)
LGL,(X2+30)	Same as above
LGL,$Z(X2+30)	Same as above
PSE,96	Turn off all sense lights
MSE,100	Test sense light 4

Class 3. Load and Store Index Operations

General format: $\not{O}P(X_n),AD(M_i)$

where: "X_n" represents the operand index reigster

$AD(M_i)$ = $NAME(M_0M_1M_4M_6)$,$NAME_n(M_0M_1M_4)$,$\not{\$}CS(M_0M_1)$,

$\not{\$}CS_n(M_0M_1)$,$\not{\$}D(M_0M_1)$,$\not{\$}D_n(M_0M_1)$,$\not{\$}L_n(M_0)$,

$\not{\$}Z(M_0M_1M_4)$,$Ln(M_0M_1M_4)$,$F(M_0M_1M_4)$,$FS(M_0M_1M_4M_6)$,$^*(M_0M_1)$.

OPERATIONS IN CLASS 3

MNEMONIC	MEANING
LAC	Load complement of address in index
LDC	Load complement of decrement in index
LXA	Load index from address
LXD	Load index from decrement
PAC	Place complement of address in index
PAX	Place address in index
PDC	Place complement of decrement in index
PDX	Place decrement in index
PXA	Place index in address
PXD	Place index in decrement
SXA	Store index in address
SXD	Store index in decrement

EXAMPLES:

EXAMPLE	MEANING
LXD(X1),NAME(ØWP)	Count of control word to X1
LAC(X2),NAME(ØWP)	Complement of control word address to X2
PXD	Clear accumulator
PAX(X3)	Address of accumulator to X3

Class 4. Tape Manipulation Operations

General format: ØP, AD(M_i)

where: AD(M_i) = N(M_0M_3),ØZ($M_2M_3M_4M_5$).

OPERATIONS IN CLASS 4

MNEMONIC	MEANING
BSFA	Backspace file, channel A
BSFB	Backspace file, channel B
BSRA	Backspace record, channel A
BSRB	Backspace record, channel B
REWA/B	Rewind, channel A or B
RTBA/B	Read tape binary, channel A or B

MNEMONIC	MEANING
RTDA/B	Read tape decimal, channel A or B
RUNA/B	Rewind and unload, channel A or B
SDHA/B	Set density high, channel A or B
SDLA/B	Set density low, channel A or B
WEFA/B	Write end-of-file, channel A or B
WTBA/B	Write tape binary, channel A or B
WTDA/B	Write tape decimal, channel A or B

EXAMPLES:

EXAMPLE	MEANING
SDLB,3	Set tape B3 for low density
WEFA,\$Z(A_n)	Write end-of-file on tape, address stored

Class 5. Special Sense Operations

General format: ∅P,\$Z(X_n)

OPERATIONS IN CLASS 5

MNEMONIC	MEANING
CHS	Change sign
CLM	Clear magnitude
C∅M	Complement magnitude
DCT	Divide check test
EFTM	Enter floating trap mode
ESTM	Enter select trap mode
ETM	Enter trapping mode
FRN	Floating round
LBT	Low-order bit test
LFTM	Leave floating trap mode
LTM	Leave trapping mode
PBT	P-bit test
RCT	Restore channel traps
RND	Round
SSM	Set sign minus
SSP	Set sign plus

Warning note: Care should be used in entering any of the trapping modes, since IVY occupies lower core; and attempts to set up trapping routines there may destroy parts of IVY.

Class 6. Convert Operations

General format: ØP,AD(M_i),N

where AD(M_i) = NAME($M_0M_1M_2M_3M_4M_5$),NAME$_n$($M_0M_1M_2M_3M_4M_5$),

$Z($M_1M_2M_3M_4M_5$),$CS/CS_n$($M_0M_1M_2M_3$),

$D/$D$_n$($M_0M_1M_2M_3$).

Note: These operations are not indexable. Xn can be only 0 or X1, and represents a special operation. See 7090 manual.

OPERATIONS IN CLASS 6

MNEMONIC	MEANING
CAQ	Convert by addition from MQ
CRQ	Convert by replacement from MQ
CVR	Convert by replacement from AC

Class 7.

This class consists of the single operation ENB, Enable from Y.

The format is

ENB(I),AD(M_i)

where "(I)" is entered if indirect addressing is desired. The allowed "AD(M_i)" are the same as for class 1 except that modifiers M_6 and M_7 are not allowed, and the form "N(M_0M_3)" is not allowed.

Class 8. Transfer and execute operations

General format: ØP(I),AD(M_i)

where "(I)" is entered if indirect addressing is desired;

AD(M_i) = $Z($M_0M_1M_2M_3M_4M_5$),L$_n$($M_0M_1M_2M_3M_4M_5$),

F/FS($M_0M_1M_2M_3M_4M_5$),*($M_0M_1M_2M_3M_4M_5$).

OPERATIONS IN CLASS 8

MNEMONIC	MEANING
HTR	Halt and transfer
TCH	Transfer in channel
TCNA/B	Transfer on channel A/B not in operation
TCØA/B	Transfer on channel A/B in operation
TEFA/B	Transfer on end-of-file, channel A/B
TIF	Transfer if indicators off
TIØ	Transfer if indicators on
TLQ	Transfer on MQ less
TMI	Transfer on minus
TNØ	Transfer on no overflow
TNZ	Transfer on non-zero
TØV	Transfer on overflow
TPL	Transfer on plus
TQØ	Transfer on MQ overflow
TQP	Transfer on MQ plus
TRA	Transfer
TRCA/B	Transfer on redundancy check, channel A/B
TTR	Trap transfer
TZE	Transfer on zero
XEC	Execute

EXAMPLES:

EXAMPLE	MEANING
TNZ,*(3)	If accumulator is not zero, transfer to third instruction following
TCØA,*	If channel A in operation, transfer to this location
TPL,(X3+3)	Return to third word of calling sequence if accumulator is plus
TRA,L2(5)	Transfer to fifth instruction following "L2" entry

Class 9. Indicator operations

General format: ØP,AD(M_i)

where AD(M_i) = $\not{\text{Z}}$($M_0M_1M_4$),N. (18 bit address)

OPERATIONS IN CLASS 9

MNEMONIC	MEANING
IIL	Invert indicators, left half
IIR	Invert indicators, right half
LFT	Left half indicators, off test
LNT	Left half indicators, on test
RFT	Right half indicators, off test
RIL	Reset indicators, left half
RIR	Reset indicators, right half
RNT	Right half indicators, on test
STL	Set indicators, left half
SIR	Set indicators, right half

EXAMPLES:

Assume GE = B(777777),TH = B(110623)

EXAMPLE	MEANING
RNT,(GE)	Skip next instruction if all right indicators are on
SIL,(TH)	Set left indicators to 110623_8
SIR,125713	Set right indicators to 125713_{10} or 365421_8

Class 10. Class 10 consists of the single operation "STA". This instruction has the same format as those in class 1, except that the address "$A_n(M_0)$" is also allowed.

Class 11. Index testing and incrementing transfers

General format: OP(X_n),AD(M_i),D

where "X_n" is the index register operand;

AD(M_i) = L_n($M_0M_1M_4$),F/FS($M_0M_1M_4$),*($M_0M_1M_4$).On "TSX," also ∅SR

D = immediate decrement = NAME($M_0M_1M_7$), $NAME_n$(M_0M_1),

∅Z(M_0M_1),N. Decrement not allowed on "TSX".

OPERATIONS IN CLASS 11

MNEMONIC	MEANING
TIX	Transfer on index
TNX	Transfer on no index
TXH	Transfer on index high
TXI	Transfer with index incremented
TXL	Transfer on index low
TSX	Transfer and set index

EXAMPLES:

EXAMPLE	MEANING
TSX(∅P),∅LD	Pathfinder branch to "∅LD"
TXI(X1),*(1),1	Increment X1 by 1 and go to next instruction
TXL(X2),L1, NAME(∅WC)	If index X2 is less than or equal to the count of "NAME," go to "L1" entry

Class 12. Variable length operations

General format: ∅P,AD(M_i),N

where AD(M_i) = NAME ($M_0M_1M_2M_3M_4M_5$),$NAME_n$($M_0M_1M_2M_3$),

∅CS/∅CS_n($M_0M_1M_2M_3$),∅D/∅D_n($M_0M_1M_2M_3$),

∅Z($M_0M_1M_2M_3M_4M_5$),X,N(M_3)

OPERATIONS IN CLASS 12

MNEMONIC	MEANING
VDH	Variable length divide or halt
VDP	Variable length divide and proceed
VIM	Variable length multiply

Class 13. Miscellaneous operations using no address

General format: ØP,AD(M_i)

AD(M_i) are the same forms as allowed in class 1, but since addresses are not used in these instructions, no error checking is performed.

OPERATIONS IN CLASS 13

MNEMONIC	MEANING
HPR	Halt and proceed
IIA	Invert indicators from accumulator
NØP	No operation
ØAI	"Or" accumulator and indicators
PAI	Place accumulator in indicators
PIA	Place indicators in accumulator
RIA	Reset indicators from accumulator

Class 14. Input-output channel commands

General format: ØP(I),AD(M_i),CT

AD(M_i) = same as class 1, except X_n = null or X2 (for transmit or no transmit)

CT = same as decrement of class 11

OPERATIONS IN CLASS 14

MNEMONIC	MEANING
IØCD	Input-output under count control and disconnect
IØCP	Input-output under count control and proceed
IØRP	Input-output of a record and proceed
IØCT	Input-output under count control and transfer
IØRT	Input-output of a record and transfer
IØSP	Input-output until signal and proceed
IØST	Input-output until signal and transfer

Coding example:

```
L|WTBA,3:RCHA,L1:TCØA,*:...:(L1)IØCT,NAME,NAME($WC)
```

The block "NAME" is written on tape 3.

Class 15. Miscellaneous sense and input-output operations

General format: ∅P,(X_n)

OPERATIONS IN CLASS 15

MNEMONIC	MEANING
BTTA/B	Beginning of tape test, channel A/B
ENK	Enter keys
ETTA/B	End of tape test, channel A/B
I∅T	Input-output check test
RCD	Read card reader, channel A
RDCA/B	Reset data channel A/B
RPR	Read printer, channel A
SLF	Turn sense lights off
SLNY	Turn on sense light Y(Y=1,2,3, or 4)
SLTY	Test sense light Y(Y=1,2,3, or 4)
SWTZ	Test sense switch Z(Z=1,2,3,4,5, or 6)
WPB	Write printer binary, channel A
WPD	Write printer decimal, channel A
WPU	Write punch, channel A

Coding Example:

To restore paper in channel A printer, assuming Share 2 board:

```
L|WPD:SPR,1:TC∅A,*:...
```

Class 16. Exchange operations

General format: ∅P

OPERATIONS IN CLASS 16

MNEMONIC	MEANING
XCA	Exchange contents of accumulator and MQ
XCL	Exchange logical contents of accumulator and MQ

Class 17. Immediate index loading operations

General format: $\not{O}P(X_n)$,$AD(M_i)$

where "X_n" is the operand index register

$AD(M_i)$ = immediate address = $NAME(M_0M_1M_6M_7)$,

$AD_n(M_0M_1)$,$\not{\$}Z(M_0,M_1)$,N

OPERATIONS IN CLASS 17

MNEMONIC	MEANING
AXT	Address to Index True
AXC	Address to Index Complemented

Coding example: To set a block to zero

```
L|AXC(X1),NAME(ØWC):STZ,NAME(X1):TXI(X1),*(1),1:TXH(X1),*(-2),0:....
```

Summary. The following table is a list of mnemonic operations in alphabetical order, giving the class to which each operation belongs. This table can be used both to find which mnemonics are allowed as well as to find to which class a given operation belongs.

TABLE A2

ALLOWED MNEMONICS AND CLASS OF OPERATIONS

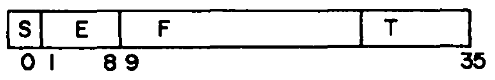
OP	CLASS	OP	CLASS	OP	CLASS	OP	CLASS	OP	CLASS	OP	CLASS	OP	CLASS	OP	CLASS
ACL	1	DVH	1	IØRP	14	MZE	1	REWA	4	SLF	15	TCØA	8	UFA	1
ADD	1	DVP	1	IØRT	14	NØP	13	REWB	4	SLNY	15	TCØB	8	UFM	1
ADM	1	EFTM	5	IØSP	14	NZT	1	RFT	9	SLQ	1	TEFA	8	UFS	1
ALS	2	ENB	7	IØST	14	ØAI	13	RIA	13	SLTY	15	TEFB	8	USM	1
ANA	1	ENK	15	IØT	15	ØFT	1	RIL	9	SLW	1	TIF	8	VDH	12
ANS	1	ERA	1	LAC	3	ØNT	1	RIR	9	SPR	4	TIØ	8	VDP	12
ARS	2	ESTM	5	LAS	1	ØRA	1	RIS	1	SPT	4	TIX	11	VLM	12
AXC	17	ETM	5	LBT	5	ØRS	1	RND	5	SPU	4	TIQ	8	WEFA	4
AXT	17	ETTA	15	LCHA	1	ØSI	1	RNT	9	SSM	5	TMI	8	WEFB	4
BSFA	4	ETTB	15	LCHB	1	PAC	3	RPR	15	SSP	5	TNØ	8	WPB	15
BSFB	4	FAD	1	LDC	3	PAI	13	RQL	2	STA	10	TNX	11	WPD	15
BSRA	4	FAM	1	LDI	1	PAX	3	RTBA	4	STD	1	TNZ	8	WPU	15
BSRB	4	FDH	1	LDQ	1	PBT	5	RTBB	4	STI	1	TØV	8	WTBA	4
		FDP	1												
BTTA	15	FMP	1	LFT	9	PDC	3	RTDA	4	STL	1	TPL	8	WTBB	4
BTTB	14	FRN	5	LFTM	5	PDX	3	RTDB	4	STØ	1	TQØ	8	WTDA	4
CAL	1	FSB	1	LGL	2	PIA	13	RUNA	4	STP	1	TQP	8	WTDB	4
CAQ	6	FSM	1	LGR	2	PSE	2	RUNB	4	STQ	1	TRA	8	XCA	16
CAS	1	HPR	13	LLS	2	PXA	3	SBM	1	STR	1	TRCA	8	XCL	16
CHS	5	HTR	8	LNT	9	PXD	3	SCHA	1	STT	1	TRCB	8	XEC	8
CLA	1	IIA	13	LRS	2	PZE	1	SCHB	1	STZ	1	TSX	11	ZET	1
										SUB	1				
CLM	5	IIL	9	LTM	5	RCD	15	SDLA	4	SWTZ	15	TTR	8		
CLS	1	IIR	9	LXA	3	RCHA	1	SDLB	4	SXA	3	TXH	11		
CØM	5	IIS	1	LXD	3	RCHB	1	SDHA	4	SXD	3	TXI	11		
CRQ	6	IØCD	14	MPR	1	RCT	5	SDHB	4	TCH	8	TXL	11		
CVR	6	IØCP	14	MPY	1	RDCA	15	SIL	9	TCNA	8	TZE	8		
DCT	5	IØCT	14	MSE	2	RDCB	15	SIR	9	TCNB	8	UAM	1		

Longhand coding example. The following longhand example is equivalent, on the 7090, to the "Mix cross sections" code in Chapter 4, page 78. The example should be self-explanatory when compared with the algebraic version.

```
1 2                                                      RUN      72
C MIX ^ CRØSS ^ SECTIØNS
L (MX)SXA($P),L4: * I=X1: *S=X2: *M=X3
  AXC(M),1: (L5)CLA,MS(M): PAC(S): TXL(S),L1,0
  CLA,C($WP): ADD,CXX(M): STA,A1: STA,A1: STA,A1
  AXC(I),1: (L6) STZ,C(I+A1): TXI(I),*(1),-1: TXH(I), L6,-CXX2-1
  (L2) CLA,MN(S): PAC(I): LDQ, MDV(S): FMP,EV: ADD,1.0: STØ,T1
  TM1,L3
  LDQ,MD(S): FMP,T1: SLW,T1: CLA,C($WP): ADD,CXX(I): STA,A2
  AXC(I),1: (L7)LDQ,T1: FMP,C(I+A2): FAD,C(I+A1): STØ,C(I+A1): TXI(I),*(1),-1
  TXH(I),L7,-CXX2-1
  CLA,MD(S): TMI,L1
  TXI(S),L2,-1
  (L1)TXI(M),*(1),-1: TXH(M),L5,-MM-1: CLA,ICT: TNZ,L4
  TSX($P),$PR: CSW,$F,FM1($WP): CSW,$A,C($WP): CSW,HM: CSW,GM: HTR
  (L4)AXT(X3),*: TRA,(X3+1)
  (L3)TSX($P),$ØP: CSW,EP2($WP)
```

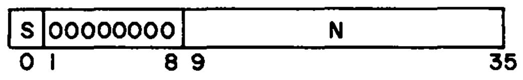
Internal formats on the 7090. The formats of internal words on the 7090 are important if the programmer is to know how to handle them using the longhand instruction set. These formats are as follows:

1. Floating point words:



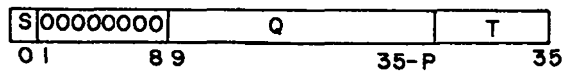
where S = the sign bit; E = exponent + 200_8; F = normalized fraction; T = tag, if any.

2a. Fixed point words:



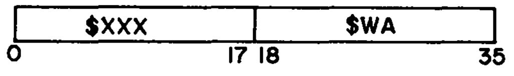
where S = the sign bit; the next eight bits are always żero; and N is the fixed point number of 27 bits or less, right adjusted so that the low order bit occupies position 35.

b. Fixed point double-stored words, tag length P:



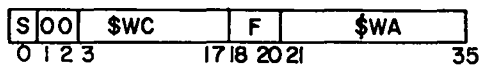
where S = the sign of "Q"; Q = the "Q" portion of 27-P bits or less, right adjusted so that the low order bit occupies position 35-P; and T = the "T" portion, of P bits or less, right adjusted so that the low order bit occupies position 35.

3. Calling sequence words:



where "$XXX" is the BCD representation of the characters "XXX", right adjusted so that the last character loaded occupies bit positions 12-17; and "$WA" represents the quantity, if any, in the "$WA" portion, right adjusted so that the low order bit is in position 35.

4. Control words, in symbol table <u>and</u> calling sequences:



where "S" is the sign bit, minus if "$\not{}$WA" gives the the address of a table of control words (e.g. for numbered symbols) or of a particular control word (renamed blocks); "∅WC" is the count of the block; "F" is the control word flag, having values as follows:

F	MEANING
0	data block
1	longhand code block
2	algebraic code block
3	remark block
4	index register symbol
5	not used
6	calling sequence block
7	not yet defined as to content

"∅WA" is the address of a control word or of a table of control words, or the base address of the named block minus one.

APPENDIX 3

THE 7030 LONGHAND INSTRUCTION SET

Like the 7090 instruction set, the 7030 set is divided into classes according to the types of addresses allowed. As a general rule these classes follow the divisions of 7030 operations according to type, as well. The reader is referred to the IBM Reference Manual, 7030 Data Processing System, for descriptions of the operations and coding examples. The organization of this appendix will follow that of Appendix 2 as closely as possible.

Addressing conventions. In the description of each class of operations, the symbols "AD," "AE," etc., are used to represent the allowed address symbols for the particular class. Each class may use a subset of the following set of address symbols:

SYMBOL	MEANING
NAME	The symbolic name of a data, remark, or calling sequence block. This may also be the name of a numbered block
NAME_n	The nth element of data block "NAME"
$CS	The calling sequence block.

SYMBOL	MEANING
$\cancel{S}CS_n$	The nth element of the calling sequence block
$\cancel{S}D$	Any of the subroutine data blocks
$\cancel{S}D_n$	The nth element of any of the subroutine data blocks
A_n	Store address symbol
$\cancel{S}L_n$	One of the elements of the "$\cancel{S}L$" block
$\cancel{S}Z$	Zero or null address
L_n	Name of an "L" entry
F	Formula name
FS	Formula set name
$\cancel{S}SR$	Name of an IVY subroutine
*	Location of current instruction
X	Fixed or floating point literal (decimal only)
N	Absolute address equal to the number N
P	Absolute address equal to the value of parameter algebra P
X_n	Address of nth index register
$\cancel{S}P$	Address of the pathfinder register
$\cancel{S}S_n$	Address of system symbol n

System symbols. On the 7030, the special registers such as the accumulator, factor register, etc., are addressable, and the "$\cancel{S}S_n$" symbols provide the ability to address these registers in any instruction.

Table A3.1 lists the system symbols, their meaning, and the absolute address which will be assembled when these symbols are specified. The address is given in the form "word address.bit address". In instructions which have less than a 24-bit address field, some of these addresses must be truncated to the nearest half or full word address to fit the instruction concerned.

TABLE A3.1

SYSTEM SYMBOLS USED IN 7030 LONGHAND

SYMBOL	MEANING	ADDRESS
$S0	location of zero	0.0
$S1	interval timer	1.0
$S2	time clock	1.28
$S3	interrupt address	2.0
$S4	upper boundary	3.0
$S5	lower boundary	3.32
$S6	boundary control bit	3.57
$S7	maintenance bits	4.0
$S8	channel address	5.12
$S9	other CPU	6.0
$S10	left zeros counter	7.17
$S11	all ones counter	7.44
$S12	left accumulator	8.0
$S13	right accumulator	9.0
$S14	sign byte register	10.0
$S15	indicator register	11.0
$S16	mask register	12.0
$S17	remainder register	13.0
$S18	factor register	14.0
$S19	transit register	15.0

In addition, the index registers can be directly addressed by using the symbols "X_n" (X0,X1,...,X15) outside parentheses.

Address modifiers. In each class of operations, address modifiers (which occur within parentheses following the address symbol) are represented by "M_i", and each symbol is allowed a subset of the following set of modifiers:

M_i	MEANING
M_0	no modifier
M_1	P (parameter algebra)
M_2	X_n + P (parameter algebra plus index register)
M_3	X_n (index register)
M_4	A_n (stored address)
M_5	$X_n + A_n$ (index register plus stored address)
M_6	$WP (control word position
M_7	$WC/$WA (control word count or address)

If parameter algebra appears as a modifier (M_1,M_2), the result of the parameter algebra modifies an address field appropriate to the instruction, e.g., 18 bits in floating point, 19 bits in branches, and 24 bits in VFL. In the description of operations by classes, each description contains notes on how parameter algebra modifiers are handled in the particular class.

Operation modifiers. Certain classes of operations may be followed by modifiers, which directly modify the operation itself. Operation modifiers "$\emptyset_i$" follow the operation mnemonic enclosed in parentheses. These modifiers are as follows:

ϕ_i	MEANING
ϕ_0	no modifier
ϕ_1	N/U (normalized or unnormalized)
ϕ_2	M (minus modifier)
ϕ_3	MA (minus absolute modifier)
ϕ_4	A (absolute modifier)
ϕ_5	MD,F,B (MD = mode = B,BU,D, or DU; F and B = parameter algebra)
ϕ_6	I (immediate modifier)
ϕ_7	V±I/V±IC/V±ICR (progressive indexing modifiers)
ϕ_8	bbbb (four bit binary connective)
ϕ_9	IND(indicator mnemonic — see Class 9 for list).
ϕ_{10}	I/B/BI (immediate, backwards, and backwards immediate modifiers)
ϕ_{11}	X_n/P(index register or parameter algebra count for transmits)

Longhand instructions are separated from one another by colons. The translation to machine language is strictly one-to-one, i.e., each entry between colons corresponds to exactly one machine instruction (which may be a full or half word). The elements of an instruction are separated from one another by commas and sets of closed parentheses. The general format of a longhand instruction is:

$$:\emptyset P(\emptyset_i)\ldots(\emptyset_\ell),AD(M_j),AE(M_\ell):$$

Conventions are explained in the discussion of each class.

General form of a longhand code. Longhand instructions are entered on "L" cards, which may be intermixed with discretion in any desired manner with "I" cards in order to supplement the algebraic code in as great or small a manner as desired. However, in this appendix we shall consider longhand formula sets, i.e., complete routines or subroutines coded in longhand. The programmer interested only in inserting a card or two of longhand code into his program should have no difficulty once the larger concepts have been studied.

Longhand codes are divided into formula sets, formulas, and local entries just as are algebraic codes. The same rules for referencing discussed in Chapter 5 also apply to longhand codes. However, the methods of forming entry points and branches of course differ between the two systems. Formula set names, formula names, and L-entries precede the operation at the entry point in parentheses, thus:

$$(FS)\emptyset P,AD(M_i)$$

$$(F)\emptyset P,AD(M_i)$$

$$(L_n)\emptyset P,AD(M_i)$$

In branches to these entries, the name of the entry occurs in the address of the instruction outside parentheses, modified if necessary, e.g.,

$$B,FS(M_i)$$

$$B,F(M_i)$$

and so on for other types of branch instructions.

Subroutine conventions. Internal IVY subroutines or subroutines coded in the algebraic language must be entered by the two instructions

```
LVI($P),*:B,NAME
```

where "$P" is the pathfinder register and "NAME" is the name of the subroutine. On the 7030, the pathfinder register "$P" is synonymous with index register X15. Thus the following entry is equivalent to the above:

```
LVI(X15),*:B,NAME
```

If a subroutine written in longhand is entered by a pathfinder branch from a portion of the code in algebraic language, or by means of one of the above instructions, one can use the first instruction of the subroutine to store the pathfinder contents in some specified location, if desired, thus:

(NAME)SV($P),AD($M_i$) or (NAME)SV(X15),AD($M_i$)

Calling sequences. Calling sequences may be constructed after a pathfinder branch to any subroutine by the use of the pseudo-operation "CSW," as in 7090 longhand; for example:

```
L|LVI($P),*:B,NAME:CSW,$ABC,GE($WP):CSW,$CT2,CRA($WC):CSW,1+TH:...
```

The IVY subroutines with variable length calling sequences (namely $AP, $TP,$OD,$PR,$PH,$MP,$DK, and $CM) require a "CSW" with no fields following it after the last calling sequence word containing some sort of entry. The compiler automatically inserts this zero word in calling sequences to these routines written in the algebraic code, but this is not done automatically in the longhand system because, as mentioned previously, longhand code is translated in a one-to-one fashion. Thus, for instance, to print a vector, the following statements should be used:

```
L|LVI($P),$:B,$PR:CSW,$F,FØRMAT($WP):CSW,VECTØR($WP):CSW:...
```

Calling sequence words and pathfinder branches take up full words on the 7030 when translated. Thus, as a general rule, one should enter pathfinder branches in any alternative exits from a subroutine; this must be done when the subroutine is coded in the algebraic language. Longhand subroutines, of course, can be coded to take half word exits into account, if desired.

Definition of "$D" blocks. Definitions of "$D" blocks in longhand code must be entered between a single pair of colons immediately following the first instruction of the formula set; for example:

```
L|(FS)SV(SP,AD(M1):$D(4),$DC(3) = 2.156,3.172,5.171,
 |$DH(5,9,10),$DM(50) = 5.152,(S)47,5.213,5.215:...
```

The format for defining "$D" blocks is otherwise the same as the format for defining them on "I" cards, discussed in Chapter 5, pages 112-114. Since "$D" definitions represent instructions to the compiler, a 7030 instruction is not, in this case, assembled to correspond to the information between colons.

Renaming. One may rename an index register with an alphabetic symbol in the same manner as on "I" cards, by entering between colons as follows:

:*SYMBØL = X_n:

Again, since this entry represents an instruction to the compiler, no information is assembled.

Storing addresses. The "A_n" symbols may be used in the address fields of "store value in address" instructions, and the address being stored will be placed in instructions having addresses modified by this same "A_n" symbol. In longhand coding, instead of using "$A_{n.m}$" to specify the number m of "store value in address" instructions, one must instead write down all "m" instructions needed. For example:

```
L|LV(XO),FRN($W):V+(XO),FRNX(X1):SVA(XO),A1:SVA(XO),A1:SVA(XO),A1
 |...LFT(N),FRN(X2+A1):*+(N),FRN(X2+A1):...:ST(N),FRN(X2+A1):...
```

However, it should be noted immediately that the above technique is not recommended. The "SVA" instruction is very slow, and when several are used in sequence as above, even slower. Instead, some temporary location and a free index register should be used to simulate the "SVA". This is what IVY actually does in the translation of such a code written in Algebraic language. Thus the above is better written as follows:

```
L|L(U),FRN(SW):+(U),FRNX(X1):ST(U),D1:...
 |...LV(X14),D1(1):LVS(X14),X2,X14:LFT(N),$Z(X14):*+(N),$Z(X13):...:
 |    ST(N),SZ(X14):...
```

where "D1" is some suitably chosen temporary location. Note that unnormalized floating point is always used in address computation; all so-called "fixed point" numbers are always stored and operated upon as unnormalized floating point, with an exponent of zero and the low order bit in bit position 49. Also note that the effective address of instructions with a stored address is computed by placing the sum of the computed address and of the modifying index register into a spare index register, and placing an address of zero (ØZ) in the instructions with "stored" addresses. This technique should always be used; if possible, "SVA" should never appear in a code.

Further restrictions on longhand coding.

1. Only fifteen index registers (X1,X2,...,X15) are allowed as modifiers in 7030 longhand coding. In addition, however, index zero, X0, is allowed as an operand in indexing operations or in the address field of other operations.

2. The programmer should note that X15 is used not only to simulate the pathfinder register, but also to simulate X13, X14, X15, X16, etc., in the algebraic code. (X13 and X14 are used to simulate the store address in algebraic code, and for other temporary purposes). Thus, if a longhand code consists only of a few cards inserted in the middle of the algebraic code, the programmer must save the contents of X15, or the contents of the last index register above X12 to which he has referred, to ensure that its previous contents, if needed after the longhand segment, are not destroyed. This can be done in two ways:

   a. In the algebraic code:

      I|...,T1 = X_n

      L|... (Longhand Code)...

      I|X_n = T1,...

      where "X_n" is the last index register above X12 referred to by the algebraic code, and "T1" is some suitably chosen temporary location; or

   b. In the longhand code:

```
I|...(Algebraic code)

L|SX(X15),T1:...(Longhand code)

L|...:LX(X15),T1:...

I|...(Algebraic code)
```

the previous contents of X15, simulating some higher index register, are here preserved by the longhand code.

The coder should note that because of the difficulties involved in simulating extra index registers in algebraic programs and of properly handling "L" type branches between algebra and longhand, great care must be exercised if a short longhand insert is placed in the midst of an algebraic code. The unwary programmer should therefore attempt to follow these rules:

a. In a short longhand insert in algebraic code, index registers X13 and above should not be used, and "L" branches between algebra and longhand should be studiously avoided.

b. Generally, longhand inserts should be comprised at least of formulas so that the above-mentioned difficulties will not arise.

3. If IVY is being run on the 7030 under MCP, the programmer is not allowed to use longhand instructions relating to input-output.

4. An IVY 7030 longhand instruction, together with its address, modifiers, etc., must be complete on one card. For all practical purposes, the end of the card is treated as a colon. Thus, the last instruction on a card need not be followed by a colon.

THE IVY 7030 LONGHAND INSTRUCTIONS

Class 1. Floating Point Operations

General format: $\emptyset P(\emptyset_1)(\emptyset_0\emptyset_2\emptyset_3\emptyset_4)^*$,AD($M_i$)

$\emptyset_1$, or the mode (N or U) must always be present.

$\emptyset_2$ and $\emptyset_3$ are not allowed in some operations, noted below.

*This notation means that any one of the modifiers shown may be present.

$AD(M_i)$ = $NAME(M_0M_1M_2M_3M_4M_5M_6)$, *$NAME_n(M_0M_1M_2M_3M_4M_5)$,

$\not{\$}CS|\not{\$}CS_n|\not{\$}D|\not{\$}D_n(M_0M_1M_2M_3M_6)$, $\not{\$}L_n(M_0)$, $\not{\$}Z(M_0M_1M_2M_3M_4M_5)$,

$L_n(M_0M_1M_2M_3)$, $F(M_0M_1M_2M_3)$, $FS(M_0M_1M_2M_3M_6)$,

$*(M_0M_1M_2M_3M_4M_5)$, X, $N(M_3)$, $X_n(M_0M_1M_2M_3)$,

$\not{\$}S_n(M_0M_1M_2M_3)$.

Parameter algebra in parentheses modifies full word address (18 bits).

In E ± I, SHFL, and SHFR, i.e., the floating point immediate operations, the forms of address allowed are

$P(M_0M_3)$, $N(M_0M_3)$, X, and $\not{\$}Z(M_0M_1M_2M_3M_4M_5)$.

The addresses of these instructions are computed and inserted as 11 bits plus sign. In these cases, "P" in parentheses modifies the 11 bit field.

OPERATIONS IN CLASS 1

MNEMONIC	MEANING	"X" IF $\not{0}_2,\not{0}_3$ ALLØWED
+	Add	
+MG	Add to magnitude	
-	Subtract	
-MG	Subtract from magnitude	
*	Multiply	X
*+	Multiply and add	
*-	Multiply and subtract	
/	Divide	X
D+	Add double	
D+MG	Add double to magnitude	
D-	Subtract double	
D-MG	Subtract double from magnitude	
DL	Load double	X
DLWF	Load double with flag	X

*This notation means that any one of the modifiers shown may be present.

MNEMONIC	MEANING	"X" IF ϕ_2, ϕ_3 ALLØWED
D*	Multiply double	X
D/	Divide double	X
E+	Add to exponent	
E+I	Add immediate to exponent	
E-	Subtract from exponent	
E-I	Subtract immediate from exponent	
F+	Add to fraction	
F-	Subtract from fraction	
K	Compare	X
KMG	Compare magnitude	X
KR	Compare for range	X
KMGR	Compare magnitude for range	X
L	Load	X
LFT	Load factor register	X
LWF	Load with flag	X
M+	Add to memory	
M+MG	Add magnitude to memory	
M-	Subtract from memory	
M-MG	Subtract magnitude from memory	
R/	Reciprocal divide	X
SHF	Shift fraction	X
SHFL	Shift fraction left	
SHFR	Shift fraction right	
SLØ	Store low order	X
SRT	Store root	X
SRD	Store rounded	X
ST	Store	X

EXAMPLES:

EXAMPLE	MEANING
+(N),5.9632-06	Add indicated literal to contents of accumulator
LFT(N),6(X3)	Load factor register with C(6+VF[X3])
E+(N),CNT3(X1)	Add EXP(CNT3+C[X1]) to exponent of accumulator
E-I(U),16	Subtract 16 from exponent of accumulator
E-I(U),$Z(16)	Same as above. Note alternative form
E-I(U),(16)	Same as above. Note alternative form

Class 2. Variable Field Length (VFL) Instructions

General format: $\emptyset P(\emptyset_5)(\emptyset_0\emptyset_6\emptyset_7)(\emptyset_0\emptyset_2),AD(M_i),AE(M_j)$

$\emptyset_5$, the mode, field length, and byte size, must always be present

In to-memory operations, noted below, the $\emptyset_6$ or immediate modifier is not allowed

Some operations noted below do not permit the O_2 modifier

$AD(M_i)$, the memory reference, has two interpretations, depending on whether or not the $\emptyset_6$(immediate) or $\emptyset_7$(progressive indexing) modifier is specified.

1. Direct: $AD(M_i)$ = same as class 1. The P modifier in parentheses modifies bit addresses (24 bits).

2. Immediate: $AD(M_i) = NAME(M_0M_1M_2M_3,M_6M_7)$*,

   $X_n/\$S_n(M_0M_1M_2M_3),\$CS/\$D(M_0M_1M_2M_3,M_7),FS(M_0M_1M_2M_3,M_6M_7),$
   $\$Z(M_0M_1M_2M_3M_4M_5),*(M_0M_1M_2M_3M_4M_5),N(M_0M_3).$

   In all cases where $\emptyset_6$ is specified, the result of P in parentheses and of N outside parentheses is shifted properly and a sign byte is constructed, according to the mode, field length, and byte size specified. Normally when $\emptyset_6$ or $\emptyset_7$ is specified the "$Z" and "N" entries should be used to obtain immediate operands. The other entries supply addresses for those unusual cases where addresses are to be operands.

$AE(M_j)$, the offset field, $= P(M_0M_3)$ only.

OPERATIONS IN CLASS 2

MNEMONIC	MEANING	$\emptyset_6$ ALLOWED	O_2 ALLOWED
+	Add	X	
+MG	Add to magnitude	X	
-	Subtract	X	

*This notation means that the modifiers which appear must be separated by commas as indicated.

MNEMONIC	MEANING	$\emptyset_6$ ALLOWED	$\emptyset_2$ ALLOWED
-MG	Subtract from magnitude	X	
*	Multiply	X	X
*+	Multiply and add	X	
*-	Multiply and subtract	X	
/	Divide	X	X
CV	Convert		X
DCV	Convert double		X
K	Compare	X	X
KE	Compare if equal	X	X
KF	Compare field	X	X
KFE	Compare field if equal	X	X
KFR	Compare field for range	X	X
KR	Compare for range	X	X
L	Load	X	X
LCV	Load converted	X	X
LFT	Load factor register	X	X
LTRCV	Load transmit register converted	X	X
LTRS	Load transit and set	X	X
LWF	Load with flag	X	X
M+	Add to memory		
M+1	Add one to memory		
M+MG	Add magnitude to memory		
M-	Subtract from memory		
M-1	Subtract one from memory		
M-MG	Subtract magnitude from memory		
SRD	Store rounded		X
ST	Store		X

EXAMPLES:

EXAMPLE	MEANING
+(BU,6,1)(I),27,16(X1)	Add 27 to contents of accumulator, offset 16 + C(X1).
L(B,17,2)(V+IC),8(X2),FST	Load the number in location VF(X2), add 8 to VF(X2),subtract 1 from CF(X2). Offset = C(FST)
ST(BU,64),STR3	Store right accumulator in STR3
M-(B,16,1),$\emptyset$S12(16),2	Subtract a field from the right accumulator, offset 2, from field of same length starting at bit 16 in left accumulator.

Class 3. Connective Operations

General format: $\emptyset P(\emptyset_5)(\emptyset_8)(\emptyset_0\emptyset_6\emptyset_7)$,AD($M_i$),AE($M_j$)

This class is similar to class 2 (just considered) except for the following:

1. The mode specified in $\emptyset_5$ must be BU.

2. The connective modifier $\emptyset_8$ must always be present. The minus modifier $\emptyset_2$ is never allowed.

3. No sign byte is ever constructed for immediate addresses.

OPERATIONS IN CLASS 3

MNEMONIC	MEANING	$\emptyset_6$ ALLOWED
C	Connect	X
CM	Connect to memory	
CT	Connect for test	X

EXAMPLES:

EXAMPLE	MEANING
CM(BU,1,1)(1111),$S15(63)	Turn on noisy mode indicator
C(BU,64,8)(0111),ARF1,20	"Or" (inclusive) contents of ARF1 and accumulator, offset 20
C(BU,24,1)(0000),0	Clear accumulator
CT(BU,FN+3,8)(0011),PLN(64+X3)	Test indicated memory field for zero

Class 4. Direct Index Arithmetic

General format: OP(X_n),AD(M_i)

where X_n is the operand index register

AD(M_i) = same as class 1. P in parentheses may, in these operations, modify full word (18 bit) or half word (19 bit) addresses, as indicated below.

OPERATIONS IN CLASS 4

MNEMONIC	MEANING	P MODIFIES	MNEMONIC	MEANING	P MODIFIES
KC	Compare count	19 bits	SC	Store count	19 bits
KV	Compare value	19 bits	SR	Store refill	19 bits
LC	Load count	19 bits	SV	Store value	19 bits
LR	Load refill	19 bits	SVA	Store value in address	19 bits
LV	Load value	19 bits	SX	Store index	18 bits
LVE	Load value effective	19 bits	V+	Add to value	19 bits
LX	Load index	18 bits	V+C	Add to value, count	19 bits
RNX	Rename index	18 bits	V+CR	Add to value, count, and refill	19 bits

Class 5. This class is composed of the single operation "LVS," or "Load value with sum". The format of this operation is:

$LVS(X_n),X_i,X_j,\ldots,X_m$

where X_n is the operand index register and the others are the index registers the sum of whose value fields are placed in the value field of X_n.

Class 6. Immediate Index Arithmetic

General format: $\emptyset P(X_n),AD(M_i)$

where X_n is the operand index register

$AD(M_i) = NAME(M_0M_1,M_6M_7),X_n/\$S_n(M_0M_1),\$CS/\$D(M_0M_1,M_7),$

$FS(M_0M_1,M_6M_7),\$Z(M_0M_1M_4),*(M_0M_1M_4),N$

The result of "P" in parentheses and "N" may be placed in an 18-bit or 19-bit field in the instruction, as noted below. Normally the "$Z" and "N" entries are used; the other entries listed above supply address fields, in the rare cases when these are wanted as operands.

OPERATIONS IN CLASS 6

MNEMONIC	MEANING	LENGTH OF P OR N
C+I	Add immediate to count	18 bits
C-I	Subtract immediate from count	18 bits
KCI	Compare count immediate	18 bits
KVI	Compare value immediate	19 bits
KVNI	Compare value negative immediate	19 bits
LCI	Load count immediate	18 bits
LRI	Load refill immediate	18 bits
LVI	Load value immediate	19 bits
LVNI	Load value negative immediate	19 bits
V+I	Add to value immediate	19 bits
V+IC	Add to value immediate, count	19 bits
V+ICR	Add to value immediate, count, refill	19 bits
V-I	Subtract from value immediate	19 bits
V-IC	Subtract from value immediate, count	19 bits
V-ICR	Subtract from value immediate, count, refill	19 bits

Class 7. Pseudo-operations to aid in indexing

These operations are used to construct index words, value fields, etc., which can be loaded into index registers using some of the indexing instructions discussed above. Each has a distinct format, and these should be studied carefully.

PSEUDO-OPERATION	MEANING	ADDRESSES ALLOWED
VF,AD(M_i)	value field	Same as Class 6. A signed, 24-bit address is assembled at the next half-word location. "P" and "N" are 24-bit quantities.
CF,AD(M_i)	count field	Same as Class 6. An unsigned, 18-bit address is assembled at the next half-word location. "P" and "N" are 18-bit quantities.

PSEUDO-OPERATION	MEANING	ADDRESSES ALLOWED
RF.AD(M_i)	refill field	X_n/\$$S_n$($M_0M_1$).*($M_0M_1$), L_n(M_0M_1) F/FS(M_0M_1),\$CS/\$CS.(M_0M_1) \$D/\$$D_n$($M_0M_1$),NAME/$NAME_n$($M_0M_1$). An unsigned, 18-bit address is assumed as the neat half-word location. "P" is an 18 bit quantity.
XW AD(M_i).AE(M_i), AF(M_k),P	index word	First field: same as "VF." Second field: same as "CF." Third field: same as "RF." Fourth field: parameter algebra for flag bits, ≤ 7. A 64-bit index word is assembled at the next full-word location.

EXAMPLES OF INDEX ARITHMETIC, ETC.

EXAMPLE	MEANING
V+I(X1),8	Value field of X1 is incremented by 8 half-words
C+I(X2),8	Count field of X2 is incremented by 8
V+(X3),L1:...:(L1)VF,-8	Value field of X3 is decreased by 8 bits
XW,PRF(\$WA),PRF(\$WC),*	Index word: value field contains address of PRF, count field contains count, refill field contains address of current instruction

Class 8. Unconditional Branching, Execute, and Refill

General format: ØP.AD(M_i)

AD(M_i) = \$Z($M_1M_2M_3M_4M_5$),L_n/F/FS/*($M_0M_1M_2M_3M_4M_5$),
\$SR(in branches only, if preceded by "LVI(\$P),*")
"P" in parentheses modifies half word addresses (19 bits) except as noted.

MNEMONIC	MEANING
B	Branch
BD	Branch disabled
BE	Branch enabled
BEW	Branch enabled and wait
BR	Branch relative
CNØP	Conditional no operation (inserted only if location counter is not set at full word)
EX	Execute
EXIC	Execute indirect and count
NØP	No operation
R	Refill (P modifies 18 bits)
RCZ	Refill on count zero (P modifies 18 bits)

EXAMPLES:

EXAMPLE	MEANING
B,*(5)	Branch to fifth half word following
BR,(5)	Branch to fifth half word following

Class 9. Indicator Branching

General format: $ØP(Ø_9)$, $AD(M_i)$

$Ø_9$ is one of the indicator mnemonics listed below, or its decimal numerical equivalent.

TABLE A3.2

INDICATOR MNEMONICS AND DECIMAL EQUIVALENTS

MNEMONIC	DECIMAL	MEANING
AD	16	Address invalid
AE	61	Accumulator equal
AH	62	Accumulator high
AL	60	Accumulator low
BTR	39	Binary transit
CBJ	8	Channel busy reject
CPUS	5	CPU signal
CS	13	Channel signal
DF	20	Data fetch
DS	19	Data store
DTR	40	Decimal transit
EE	11	End exception
EK	3	Exchange control check
EKJ	6	Exchange check reject
EØP	12	End of operation
EPGK	9	Exchange program check
EXE	18	Execute exception
IF	21	Instruction fetch
IJ	2	Instruction reject
IK	1	Instruction check
IR	25	Imaginary root
LC	22	Lost carry
LS	26	Lost significance
MK	0	Machine check
MØP	55	To-memory operation
NM	63	Noisy mode
ØP	15	Operation code invalid
PF	23	Partial field
PGO-PG6	41-47	Program indicators 0-6
PSH	27	Preparation shift greater than 48
RGZ	58	Result greater tnan zero
RLZ	56	Result less than zero
RN	59	Result negative
RU	34	Remainder underflow
RZ	57	Result zero
TF	35	T flag
TS	4	Time signal
UF	36	U flag
UK	10	Unit check

TABLE A3.2 (continued)

MNEMONIC	DECIMAL	MEANING
UNRJ	7	Unit not ready reject
USA	17	Unended sequence of addresses
VF	37	V flag
XCZ	48	Index count zero
XE	53	Index equal
XF	38	Exponent flag
XH	54	Index high
XL	52	Index low
XPFP	28	Exponent flag positive
XPH	30	Exponent high
XPL	31	Exponent low
XPØ	29	Exponent overflow
XPU	32	Exponent underflow
XVGZ	51	Index value greater than zero
XULZ	49	Index value less than zero
XVZ	50	Index value zero
ZD	24	Zero divisor
ZM	33	Zero multiply

For Class 9, AD(M_i) = same as Class 8, except only X1 can participate in modification.

OPERATIONS IN CLASS 9

MNEMONIC	MEANING
BI	Branch on indicator
BIZ	Branch on indicator and set to zero
BZI	Branch on zero indicator
BZIZ	Branch on zero indicator and set to zero

EXAMPLE:

BI(NM),L3 (or BI(63),L3): Go to L3 if "NM" indicator is on

Class 10. Bit Branching

General format: $\phi P, AD(M_i), AE(M_j)$

$AD(M_i)$ = same as Class 2 (direct)

$AE(M_j)$ = same as Class 9

OPERATIONS IN CLASS 10

MNEMONIC	MEANING
BB	Branch on bit
BB1	Branch on bit and set to one
BBN	Branch on bit and negate
BBZ	Branch on bit and set to zero
BZB	Branch on zero bit
BZB1	Branch on zero bit and set to one
BZBN	Branch on zero bit and negate
BZBZ	Branch on zero bit and set to zero

EXAMPLE:

BB1,$S15(63),*(2): turn on noisy mode indicator and go to next instruction

Class 11. Index Branching

General format: $\phi P(X_n)$, $AD(M_i)$

where X_n is the operand index register;

$AD(M_i)$ = same as Class 9

OPERATIONS IN CLASS 11

MNEMONIC	MEANING
CB	Count and branch
CB+	Count, branch, and increment value by full word
CB-	Count, branch, and decrease value by full word
CBH	Count, branch, and increment value by half word
CBR	Count, branch, and refill
CBR+	Count, branch, refill and increment value by full word
CBR-	Count, branch, refill, and decrease value by full word

MNEMONIC	MEANING
CBRH	Count, branch, refill, and increment value by half word
CBZ	Count and branch on zero
CBZ+	Count and branch on zero and increment value by full word
CBZ-	Count and branch on zero and decrease value by full word
CBZH	Count and branch on zero and increment value by half word
CBZR	Count, branch on zero, and refill
CBZR+	Count, branch on zero, refill, and increment value by full word
CBZR-	Count, branch on zero, refill, and decrease value by full word
CBZRH	Count, branch on zero, refill, and increment value by half word

EXAMPLE:

```
L|LX(X1),L1:Z,(X1):CB+(X1),*(-1):...:(L1)XW,BXT($WA),BXT($WC):...
```

Set block "BXT" to zero

Class 12. Data Transmission Operations

General format: $\phi P(\phi_{11})(\phi_0\phi)$,AD($M_i$),AE($M_j$)

$\phi_{11} = X_n$ in direct operations, $P \leq 15$ in immediate operations

AD(M_i),AE(M_j) = same as Class 1

OPERATIONS IN CLASS 12

MNEMØNIC	MEANING
T	Transmit
SWAP	Swap

EXAMPLE:

T(15)(I),X1,NAME: Save all 15 index registers in block "NAME"

Class 13. Two Miscellaneous Operations

General format: ϕP,AD(M_i)

AD(M_i) = same as Class 1. P modifies 18 or 19 bits as noted below

OPERATIONS IN CLASS 13

MNEMONIC	MEANING
SIC	Store instruction counter if ... (may precede half word branch and indicator branch only) (P modifies 19 bits)
Z	Store zero (P modifies 18 bits)

The following table summarizes all 7030 longhand instructions for quick reference purposes.

TABLE A2.3

7030 LONGHAND MNEMONICS AND CLASSES

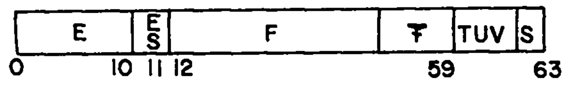
MNEMONIC	CLASS	MNEMONIC	CLASS	MNEMONIC	CLASS	MNEMONIC	CLASS
±	1,2	D*	1	LTRS	2	SWAP	12
±MG	1,2	D/	1	LV(I)	4(6)	SX	4
*	1,2	DCV	2	LVNI	6		
*	1,2	DCV	2	LVE	4	T	12
*±	1,2	DL	1	LVS	5	V+	4
/	1,2	DLWF	1	LWF	1,2	V±I	6
B	8	E±(I)	1	LX	4	V+C	4
BB(1,N,Z)	10	EX	8	M±	1,2	V±IC	6
BD	8	EXIC	8	M±MG	1,2	V+CR	4
BE	8	F±	1	M±1	2	V±ICR	6
BEW	8	K	1,2	NØP	8	VF	7
BI(Z)	9	KC(I)	4,6	R	8	XW	7
BR	8	KE	2	R/	1	Z	13
BZB(1,N,Z)	10	KF	2	RCZ	8		
BZI(Z)	9	KFE	2	RF	7		
C	3	KFR	2	RNX	4		
C±I	6	KMG	1	SC	4		
CB(+,-,H)	11	KMGR	1	SHF(L,R)	1		
CBR(+,-,H)	11	KR	1,2	SIC	13		
CBZ(+,-,H)	11	KV(I)	4(6)	SLØ	1		
		KVNI	6				
CBZR(+,-,H)	11	L	1,2	SR	4		
CF	7						
CM	3	LC(I)	4(6)	SRD	1,2		
CNØP	8						
CT	3	LCV	2	SRT	1		
CV	2	LFT	1,2	ST	1,2		
D±	1	LR(I)	4(6)	SV	4		
D±MG	1	LTRCV	2	SVA	4		

<u>Coding example</u>. The following example is equivalent to the algebraic formula "MX" discussed in Chapter 4, page 78, and should be studied thoroughly:

```
IVY  DATE    PAGE    NAME                    PROBLEM          73 74 75 76 77 78 79 80
                                                              M  X
1 2                                                               RUN            72
C MIX ^ CRØSS ^ SECTIØNS
L (MX) LX(X9), $P: * I = X1: * S = X2: * M = X3
  LX(M), L6: (L5) LV(S), MS(M+1): BI(XVZ), L1
  L(U), C($WP): +(U), CXX(M): LV(X14), $S12(1)
  LX(I), L7: V+(I), X14: Z, (I): CB+(I), *(-1)
  (L2) LV(I), MN(S+1): L(N), MDV(S): *(N), EV: +(N), 1.0: ST(N), T1
  BI(RN), L3
  *(N)(A), MD(S): ST(N), T1: L(U), C($WP): +(U), CXX(I): LV(X13), $S12(1)
  LX(I), L7: V+(I), X14: V+(X13), I: L(N), T1: *(N), (X13)
  +(N), (I): V+I(X13), 2: CB+(I), *(-4)
  L(N), MD(S): BI(RN), L1
  V+I(S), 2: B, L2
  (L1) CB+(M), L5 : L(U), ICT: BZI(RZ), L4
  LVI($P), *: B, $PR: CSW, $F, FM1($WP): CSW, $A, C($WP)
  CSW, HM: CSW, GM: CSW
  (L4) B, (X9+2)
  (L3) LVI($P), *: B, $ØP: CSW, EP2($WP)
  (L6) XW, 1, (MM): (L7) XW, 1, (CXX2)
```

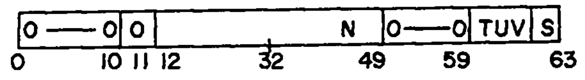
Internal formats on the 7030. The formats of internal words on the 7030 are important if the programmer is to know how to handle them using the longhand instruction set. These formats are as follows:

1. Floating point words:



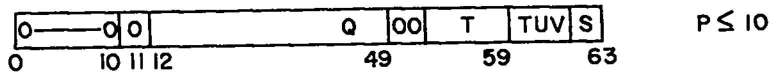
where E = exponent; ES = exponent sign; F = normalized fraction; T = tag, if any; TUV = flag bits; S = fraction sign.

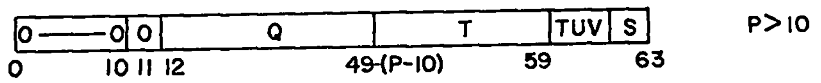
2a. Fixed point words:



where N = fixed point number of 37 bits or less, right adjusted so that the low order bit occupies position 9; TUV = flag bits; S = sign. Note that fixed point numbers are in reality floating point words, unnormalized, with exponent = 0. The number is adjusted so that it can be loaded into a value field by addressing the second half word.

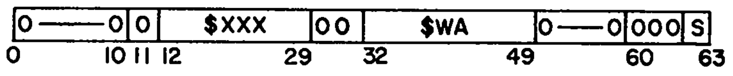
2b. Fixed point double-stored numbers, tag length P:





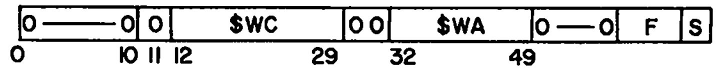
Where Q = the "Q" portion, of 37 or 37-(P-10) bits or less, right adjusted so that the low order bit appears in bit 49 or 49-(P-10); T = the "T" portion, right adjusted so that the low order bit occupies position 59; TUV = flag bits; S = the sign of "Q".

3. Calling sequence words:



where "$XXX" is the BCD representation of the character "XXX," right adjusted so that the last character loaded occupies bit positions 24-29; and "$WA" represents the quantity in the "$WA" portion, right adjusted so that the low order bit occupies position 49.

4. Control words



The quantities "$WC," "$WA," "F," and "S" are governed by the same conventions as in the 7090 control word, discussed at the end of Appendix 2.

INDEX

NOTE: Special symbols are found as follows:
"*" is found in the A's (asterisk);
"$" is found in the D's (dollar).