DRAFT

MAX 2 COMPUTER SYSTEM

REFERENCE MANUAL
and
MICROPROGRAMMING GUIDE

International Meta Systems, Inc.
3655 Torrance Blvd.   Torrance, CA 90503

MCS - 02 September 1987
MCS - 01a February 1987
MCS - 01 January 1987
Copyright International Meta Systems Inc., 1987, 1988

TABLE OF CONTENTS

SUMMARY 1

INTRODUCTION 2

I SYSTEM ARCHITECTURE 3

II PC/3230 CPU EXECUTIVE SYSTEM 8
FUNCTION KEY DEFINITIONS 13
FUNCTION KEY SUMMARY 22

PROGRAM OPERATION 23

III MICROPROGRAMMING 24
INSTRUCTION PIPELINE 26
LEFT HAND SIDE FORMAT 26
RIGHT HAND SIDE FORMAT 28
OFF-CHIP INTERFACES 29

1.0 SPECIAL-PURPOSE REGISTERS 30
1.1 X REGISTER 30
1.2 MEMORY ADDRESS REGISTER (MAR) 30
1.3 MEMORY INPUT REGISTER (MIR) 30
1.4 MEMORY OUTPUT REGISTER (MOR) 31
1.5 K REGISTER 31
1.6 LINK STACK REGISTER (LSR) 31
1.7 DATA REGISTER (DR) 31
1.8 INTERRUPT REGISTER (IR) 32
1.9 INTERRUPT MASK REGISTER (IMR) 32
1.10 EXTERNAL INPUT REGISTER (EIR) 32

2.0 INSTRUCTION FORMAT 34
2.1 "T" (TARGET OPERAND) FIELD 35
2.2 "A" (PRIMARY INPUT OPERAND) FIELD 36
2.3 OPERATOR FIELD 37
2.4 "B" (SECONDARY INPUT OPERAND) FIELD 39
2.5 OVERFLOW IN ARITHMETIC/LOGICAL/SHIFT INSTRUCTIONS 41
2.6 "C" (RIGHT HAND SIDE) FIELD 42
2.6.1 CONDITIONAL TRANSFERS, TRA ([C]=8...F) 43
2.6.2 LOAD K REGISTER, LDK ([C]=1) 45
2.6.3 UNCONDITIONAL TRANSFER, TRA ([C]=7) . 45
2.6.4 LINK INSTRUCTIONS, LINK ([C]=4) . . . 46
2.6.5 LINK CONDITIONAL, LINKC ([C]=5) . . . 46
2.6.6 RETURN INSTRUCTION, RETURN ([C]=6) . 47
2.6.7 SKIP CONDITIONAL INSTRUCTION, SKIP ([C]=0) 48
2.6.8 EXTERNAL BUS INSTRUCTIONS, BUS ([C]=2,3) 50

3.0 PIPELINE CONSTRAINTS 54
3.1 PIPELINE PHASE I 54
3.2 PIPELINE PHASE II 55
3.3 PIPELINE PHASE III 56
3.4 PIPELINE PHASE IV 57
3.5 PROGRAMMING/TIMING RULES FOR READING OR WRITING CACHE MEMORY 58
3.5.1 WRITE CACHE SEQUENCE 58
3.5.2 READ CACHE SEQUENCE 59

4.0 SPECIAL PROGRAMMING TECHNIQUES 60
4.1 LOOPS 60
4.1.1 BOTTOM TEST LOOPS 60
4.1.2 TOP TEST LOOPS 61
4.1.3 ARRAYS: MICROCACHE LOOPS 61
4.2 OUT OF SEQUENCE EXECUTION 62
4.3 TRANSFER VECTOR BRANCHING 63
4.4 REGISTER SHARING 64
4.5 ACCESSING PRESTORED DATA FROM MICROSTORE3 . 65
4.6 STACKING SUBROUTINE PARAMETERS 65

5.0 MEMORY INTERFACE 66
5.1 MEMORY CONTROLLER INTERFACE 66
5.1.1 BUS EMIT OPERATIONS 68
5.1.2 BUS RECEIVE OPERATIONS 69
5.1.3 INDEXING 72
5.1.4 TYPICAL PROGRAMMING 73

6.0 PC I/O AND SUPPORT INTERFACES 74
6.1 METAMICRO TO PC 76
6.1.1 ACCESS 77
6.1.2 STATUS TO METAMICRO 77
6.1.3 READ FUNCTIONS 78
6.1.4 WRITE FUNCTIONS 78
6.2 INTERFACE TO PC-AT 78
6.2.1 PC TO METAMICRO 79
6.2.2 ACCESS 79
6.2.3 READ STATUS FUNCTION 80
6.2.4 STATUS TO PC-AT 80
6.2.5 WRITE COMMAND REGISTER 81
6.2.6 WRITE SELECTED REGISTER 82
6.2.7 READ SELECTED REGISTER 82
6.2.8 OTHER COMMAND REGISTER FUNCTIONS . . 83

7.0 FLOATING POINT COPROCESSOR 85
7.1 BUS EMIT AND RECEIVE FUNCTIONS 85
7.2 SAMPLE PROGRAMS 87

APPENDIX A: ASSEMBLER SYNTAX AND PSEUDO OPS 89
A1 LANGUAGE ELEMENTS 90
A1.1 COMMENTS 90
A1.2 IDENTIFIERS 90

A1.3 NUMBERS 90
A1.3.1 DECIMAL INTEGERS 90
A1.3.2 HEXADECIMAL INTEGERS 90
A1.3.3 CHARACTERS 91
A1.3.4 REAL NUMBERS 91
A1.3.5 DOUBLE PRECISION NUMBERS . . . 91
A1.4 EXPRESSIONS 91
A1.4.1 LOGICAL OPERATORS 92
A1.4.2 SHIFT OPERATORS 92
A2 PSEUDO-OPERATIONS 94
A2.1 ASSEMBLER ACTION COMMANDS 94
A2.1.1 LABEL (LONG LABEL DEFINITION) . 94
A2.1.2 CHANGE, UNCHANGE (KEYWORD ALTERATION) 95
A2.1.3 INST, DATA (INSTRUCTION, DATA MODE INITIATION) 95
A2.1.4 ORG (SET ORIGIN OF ABSOLUTE CODE) 95
A2.1.5 DC (DEFINE CONSTANT) 96
A2.1.6 RS (RESERVE STORAGE) 96
A2.1.7 EQU (EQUATE SYMBOLS) 96
A2.2 ASSEMBLER LISTING COMMANDS 97
A2.2.1 HEADER (PAGE HEADINGS) 97
A2.2.2 EJECT (PAGE EJECTION) 97
A2.2.3 LIST, NOLIST (ASSEMBLY LISTING ON, OFF) 97
A2.2.4 LISTC, NOLISTC (LISTING COMMANDS ON, OFF 97
A2.2.5 BLOCKS, NOBLOCKS (BLOCK STRUCTURE ANNOTATION CONTROL) 97
A2.2.6 FORMAT, NOFORMAT (FORMATTED LISTING CONTROL) 98
A2.2.7 COMWIDTH (COMMENT WIDTH FOR JUSTIFICATION) 98
A2.2.8 ILIST, NOLIST (INCLUDE FILE PRINTING ON, OFF) 98
A2.2.9 WARN, NOWARN (WARNING MESSAGE PRINTING ON, OFF) 98
A2.2.10 END (END OF ASSEMBLY) 98
A2.2.11 CODELEN 99
A2.2.12 IFON, IFOFF 99
A2.3 DEFAULT OPTIONS 99
A3 PRE-PROCESSOR COMMANDS 100
A.3.1 #IF #ELSE #END 100
A.3.2 #INCLUDE 100
A.3.3 #DEFINE 100
A4 STANDARD ASSEMBLER MNEMONICS 101

INDEX 111

SUMMARY

This manual provides an architectural overview of the MAX 2 hardware for the PC-AT. Within that system it describes the instruction set and the symbolic assembly language used to create microprograms and explains the operating characteristics of the hardware. An Executive System, resident on the PC controls loading programs, debugging 3230 CPU programs, and the I/O between the MAX 2 and the PC. Appendix A describes assembler features and use. The descriptions of the assembly formats of the instructions are interleaved in the description of the hardware instruction set.

Italics are used in assembly-language descriptions, and tables of equivalence between assembly mnemonics and binary microcode are provided.

Assembly language descriptions employ BNF (Backus-Naur Form) to define language syntax. The BNF structure is as follows:

a. All names appearing in angle brackets, <...>, are names of syntactic types.

b. The symbol "::=" means "is defined as".

c. The symbol "|" indicates a choice.

d. Items appearing within square brackets, [...], are optional.

e. Items appearing within set braces, {...}, may be repeated zero or more times.

INTRODUCTION

The 3230 CPU is the processor element (VSLI microprocessor chip) of the MAX 2 expansion board for the IBM PC-AT and compatible personal computers. This document is a combination reference manual and microprogramming guide for the MAX 2/MetaMicro computer system. Throughout this document, micro-programming examples, used to describe the operation of the hardware, are presented in the assembly-language of the 3230 CPU.

I SYSTEM ARCHITECTURE

In recent years, two fundamental categories of computer design have been widely discussed: the conventional microcoded complex-instruction-set computer (CISC) architecture and the non-microcoded reduced-instruction-set computer (RISC) architecture, both of which rely heavily on compilers to transform application software to run on the computer. The IMS "Meta" architecture is a third category, a minimal-instruction-set computer (MISC) that is substantially different in software architecture. Meta architecture supports the execution of application programming languages without the need for conventional compilers.[1]

IMS Meta architecture resolves a bottom-up problem of VLSI performance and a top-down problem of software functionality. Both problems constrain future data processing technology, but are not resolved by either CISC or RISC approaches. Both problems can be resolved in the MISC architecture by concentration of the object program encoding, which relieves the off-chip/on-chip traffic in the so-called "von Neumann bottleneck."

Encoding concentration is facilitated by language interpretation: the direct execution of the most abstract form of a program -- a compressed image of its high-level language statements. Simple encoding algorithms compress the semantic information content of a high-level source program into a binary image that averages less than half its original size. This reduces the object program size compared to a compiled program by an order of magnitude.

VLSI Performance - The bottom-up problem of VLSI performance relates to the increasing disparity between VLSI chip speeds and the speed of off-chip circuitry such as large dynamic memories and external busses. Reducing silicon circuit feature size allows enormous gains in clock speeds of processing that takes place on-chip, roughly proportional to the reduction in VLSI feature size. Going from two-micron to one-micron feature size results in a two to threefold gain in on-chip clock speed. This magnifies the problem of the von-Neumann bottleneck, because of the large number of

---

[1] This eliminates a conventional compiler's primary function: transforming high-level language programs into expanded low-level machine code.

cycles that a processor may have to wait for information transmitted from large offchip memories.The on-chip/off-chip performance disparity can be relieved by reducing the size of the program code (instructions) and the frequency with which data must be transmitted between the very large and relatively slow dynamic memories and the faster on-chip processors and static memories. Executing a program in its most abstract (concentrated) form minimizes the instruction traffic. Using small, high-speed memory for data memorying of data structures minimizes the data traffic. The net effect is an order-of-magnitude gain in effective memory bandwidth as compared to the expanded machine-language image produced by a conventional compiler.

Software Functionality - The major top-down problem removed by the Meta approach is the complexity of high-order language implementation. Languages which have shown the highest level of programmer productivity have dynamic characteristics such as dynamic binding and data type redefinition that do not lend themselves to systems with compilers and static loaders. Application language interpretation eliminates the assembly language software interface and uses the 3230 CPU hardware instruction set to directly execute the high-level language, making the hardware into a microprogrammed igh-level language machine.

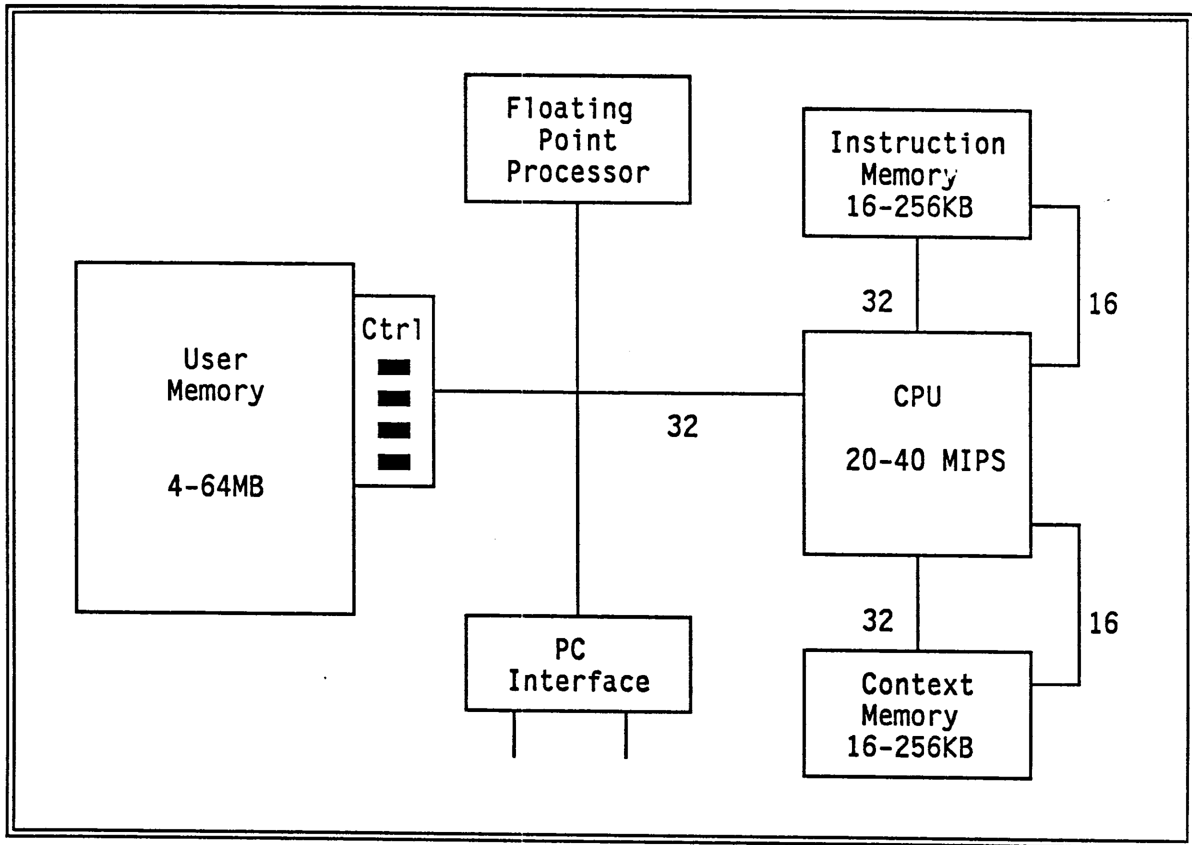


FIGURE 1 MAX 2/MetaMicro Architecture

The MAX 2/MetaMicro architecture executes compressed-source images of high-level languages via micro-programming. Functionally, it consists of the following subsystems:

A. METASYSTEM:

1. Microsystem: Minimal Instruction Set Computer (MISC)

(a) 3230 CPU Processor: a user-microprogrammable processor consisting of a single very-high-speed integrated circuit (VHSIC) chip,

(b) Instruction memory: a register-speed static memory of up to 64K 32-bit words, used as a read-only instruction memory to the 3230 CPU processor, containing interpreter loaded from the supporting PC-AT,

(c) Data memory: a register-speed static memory of up to 64K 32-bit words, used as a local read/write storage by micro-programs,

2. User memory: a large dynamic main memory external to the microsystem, used to store high-level language code and data,

3. PC-Channel: interface channel logic connecting to the computer bus of the PC-AT,

B. METACOMPUTER (Virtual Machine): a micro-program, resident in the instruction memory, that loads and encodes an application source language program into a compressed binary image in the user memory, then interprets the application language by executing the binary image. When boot-loaded from the PC, the metacomputer converts the general-purpose metasystem into a special-purpose application language machine, masking the features of the metasystem like a high-order language masks hardware features.3230 CPU's microprogramming features are optimized for the implementation of metacomputer code for any type of high-order language. Because each language has its own compressed source-executable image (instruction set architecture) that is

independent of the metasystem, the design of the metacomputer is not constrained by the design of the hardware. This offers complete freedom in the development of high-order-language machines. Since the metacomputer code resides in a separate high-speed memory (instruction memory) which is externally loadable, the hardware can change its language merely by loading a new metacomputer.

Application-language interpretation sharply reduces software complexity and leverages the performance of application languages. The computer is substantially more user-friendly because the language that it executes directly corresponds to the one in which the program was written, i.e.,the program is executed as written, not as a highly transformed machine-language program generated by a compiler.

In addition to its use as a high-level language interpretation engine, the MetaMicro is designed to function as the control element of systems such as disk controllers, telemetry processors, parallel architecture building blocks, and in other digital processing environments that use general-purpose microprogrammed logic.

The 3230 CPU performs arithmetic and logical operations on 32-bit-wide parallel data and is controlled by a microprogram employing 32-bit-wide parallel instructions. The 3230 CPU employs eight 32-bit-wide general-purpose registers and ten special-purpose registers. The 32-bitwide data memory memory of up to 64K words is directly accessible by the 3230 CPU to use as a large register file or to implement software structures such as push-down stacks and hash-addressed heap-storage. The MetaMicro may be configured to communicate with other MetaMicros or peripheral hardware via a 32-bit-wide bidirectional asynchronous external bus.

II PC/METAMICRO EXECUTIVE SYSTEM

This section describes the operation of the PC/MetaMicro development system. The system has two versions: one which runs a MetaMicro computer and a second which simulates MetaMicro execution.

The system is controlled by the user through the use of function keys. Functions are provided which:

1. Load microprograms
2. Start programs and control execution
3. Display I/O
4. Display instruction execution
5. Breakpoint at instruction location or data ref.
6. Change interpreter instruction
7. Display registers, stacks, and the external bus
8. Dump memory, data memory, registers, and stacks
9. Control the form and format of the display screen.

Information from the execution is displayed in two windows on the screen :

Upper screen window: Lines of terminal-directed input/output to and from the MetaMicro are displayed in the upper window of the PC screen. Lines of data received from the MetaMicro are scrolled upward in this window and up to 99 lines overflowing the screen are retained for subsequent browsing.

Lower screen window: A trace of the code being executed in the 3230 CPU under control of the PC is displayed in the lower window on the PC screen. Lines of data displayed are scrolled upward in this window and up to 99 lines overflowing the screen are retained for subsequent browsing.

There are two formats for the display. The normal form, shown in Figure 1, is organized into three fields: (1) The assembler -symbolic form of the executing instructions, (2) the contents of the X register in hexidecimal, and (3) the contents of the X register diplayed as characters (values <X'20' are displayed as ~).

In an alternate mode, controlled by SHIFT-F9, the lower window display is organized into three columns, from left to right:

(a) The 3230 CPU location counter - four hexadecimal characters.

(b) The 32 bit instruction of the 3230 CPU, organized left to right as the T, A, OP, F, B, C and Address fields, displayed in hexadecimal format.

(c) The X-register of the 3230 CPU organized high order (bit 31) to low order (bit 0) in binary format.

A sample of this screen display is shown in Figure 2.

```
R1:        R2:        R3:        R4:        R5:        R6:        R7:
00000002   00000003   00000004   00000005   00000000   00000000   00000000
X:         MAR:       KR:
00000AA9   0000       0AAA
STACK:  TOP -> BOTTOM
0048 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
 loc       instruction                              x register

 C6D:      MAR :=  0 = 0007F                        0000007F   ~~~~
 C6E:      R7  :=  R6 SRL 2 LDK 000                 00000000   ~~~~
 C6F:      R6  :=  R6 AND 3                         00000000   ~~~~
 C70:      MAR :=  0 = 0007E                        0000007E   ~~~~
 C71:      R7  :=  MOR - R7 TRA C95                 00000000   ~~~~
 C72:      X   :=  X = 0 TRA C73 ON NZ              00000000   ~~~~
 C95:      LSR :=  RC AND 3                         00000000   ~~~~
 C96:      KR  :=  R6 SRL 2                         00000000   ~~~~
 C97:      R7  :=  0 = 000FF                        000000FF   ~~~~
 C98:      NOP RETURN C9B                           00000000   ~~~~
 C99:      X   :=  MOR SRL 8 TRA C9A                00000000   ~~~~
 C9B:
```

waiting:

FIGURE 2

The instructions shown in figure 2, above, are shown as they move through the instruction pipeline. At the bottom of the screen the location indicates the next instructions in phase 1 and phase 2 of the execution pipeline. Phased execution is covered in Section III and shown in figure 6.

(C) Int'l Meta Systems Inc., 1986,1987

R1:	R2:	R3:	R4:	R5:	R6:	R7:
00000002	00000003	00000004	00000005	00000000	00000000	00000000
X:	MAR:	KR:	IR:			
00000006	0000	0000	00			

loc	instruction	x register
0:	1 0 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0001
1:	8 1 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0010
2:	9 1 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0011
3:	A 1 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0100
4:	B 1 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0101
5:	C 1 3 0 1 0 0 0 0	0000 0000 0000 0000 0000 0000 0000 0110
6:	D 1 3 0 1 0 0 0 0	
7:	E 1 3 0 1 0 0 0 0	
8:		

waiting:

FIGURE 3

(C) Int'l Meta Systems Inc., 1986, 1987

```
                                  Bus                  Registers                       Stack
                               bus # 00        X:   00000006 R0: 00000002        0000 0000
                               status 0000     AR:  00000006 R1: 00000003        0000 0000
                                inbus 000000   BR:  00000001 R2: 00000004        0000 0000
                               outbus 000000   MAR: 0000     R3: 00000005        0000 0000
R0:          R1:          R2                   KR:  0000     R4: 00000000        0000 0000
00000002     00000003     00                   EIR: 00000000 R5: 00000000        0000 0000
X:           MAR:         KR:        IR:       IR:  0000     R6: 00000000        0000 0000
00000006     0000         0000       00        IMR: 0000     R7: 00000000        0000 0000

loc      instruction                                  x register

C99:     X    := MOR SRL 8 TRA C9A                    00000000   ~~~~
C9B:     X    := MOR + 0 RETURN 000                   00000000   ~~~~     RUN
C9A:     R7   := X AND R7 LDK 000                     00000000   ~~~~      loc i d
76B:     R6   := X - 00022                            FFFFFFDE   ~~~~     001A *
76C:     R6   := X - 0 TRA 77F ON NZ                  FFFFFFDE   ~~~~     0000
76D:     R0   := R0 + 2                               FFFFFFDE   ~~~~     0000
76E:     R1   := X - 1                                                    go
76F:     R1   := R0 - R3 TRA 772 ON HB
76F:     R6
77F:

waiting:
```

FIGURE 4

The development system is invoked by a call for the program MMX and the interpreter by the program MMXS.

The functions supported are controlled by function keys F1 through F10 as well as the ctrl-, alt- and shift- versions of these keys. Many of the keys respond with queries for parameters. In these cases an escape or an empty line will back out of the query.

FUNCTION KEY DEFINITIONS

F1: Output Scroll Up

F1 scrolls up the MetaMicro input/output display (upper window). The last 99 lines of MetaMicro input/output are retained for display. Scrolling past the end of the saved lines yields a "no more data" message.

F2: Instruction Scroll Up

F2 scrolls up the 3230 CPU instruction display (lower window). The last 99 lines of 3230 CPU instruction execution are retained for display.

F3: Output Scroll Down

F3 scrolls down the MetaMicro input/output display (upper window).

F4: Instruction Scroll Down

F4 scrolls down the 3230 CPU instruction display (lower window).

F5: Interpreter Run Mode Toggle

F5 toggles the "run" mode. The first F5 puts the 3230 CPU into the running state. The 3230 CPU executes instructions until the next F5 terminates the "run" mode. The "run" mode permits the 3230 CPU to execute instructions independent of the controlling PC.

F6: Instruction Step Execution

F6 steps the 3230 CPU (causes the 3230 CPU to execute one cycle) and displays the results in the lower window.

F7: Clear 3230 CPU

F7 initiates a clear cycle for the 3230 CPU. The 3230 CPU's location counted is set to zero, the interrupts are reset and the pipe-line is "flushed".

F8: Set Instruction Location

Sets the location counter of the 3230 CPU. The user is prompted for a hexadecimal entry for the location counter value.

F9: Load Interpreter

F9 loads selected portions of the interpreter into the instruction memory of the 3230 CPU. The user is prompted for a starting and ending address (hexadecimal) for the portion of the memory to be loaded.

F10: Display/Edit Interpreter

F10 initiates the display and edit sequence for the interpreter of the 3230 CPU. The location of the first instruction to be displayed or edited is prompted for. The requested 3230 CPU instruction is displayed in either symbolic form or in hexadecimal format in the sequence T, A, OP, F, B, C and Address depending on the display format selected. [ENTER] moves to the next 3230 CPU instruction. Right arrow [->] moves to the first field for editing, re-displaying the fields in symbolic. Editing is accomplished by over-typing the field ([->] or [<-] leaveing it intact). An escape in any field discards any changes made to the instruction and moves to the next instruction. The cursor returns to the left of the instruction when an [ENTER] is pressed for any input field. Right arrow [->] selects a field for rediting; [ENTER] moves to the next instruction. The instruction is re-displayed in

symbolic form for correctness verification. Another F10 prompts for another location.

Modified instructions are both retained in a memory image in the PC and also written into the memory of the MetaMicro.

LOGGING AND COPYING OUTPUT TO FILES OR PRINTER

I/O to the two display windows may be directed to files or the the printer using the function keys below. When one of the function keys is pressed the system prompts the user for the name of a file to be used. Any valid DOS file name may be used or 'PRN' outputs directly to the printer. The displayed file name can be accepted with CR. Space CR selects the printer. ESC turns off logging.

ALT-F1: PC I/O Print Log Toggle

ALT-F1 toggles logging to the printer of the PC input/-output as it is displayed in the upper window. Normally, this is I/O directed to the terminal by the executing MetaMicro program, but may also include Debug output. (See Shift F1). The "L" indicator at the top of the window shows the on/off status of this switch.

ALT-F2: Interpreter Print Log Toggle

ALT-F2 toggles logging to the printer of the MetaMicro output as it is displayed in the lower window. The "L" indicator at the top of the window shows the on/off status of this switch.

ALT-F3: Print Input/Output Display Lines

ALT-F3 prints the last "n" lines of the MetaMicro input/output display (upper window). "n" is prompted for.

ALT-F4: Print Interpreter Display Lines

ALT-F4: prints the last "n" lines of the 3230 CPU instruction display (lower window). "n" is prompted for.

ALT-F5: Run-Until Address Toggle

ALT-F5 toggles the "run-until" option for the 3230 CPU. This option allows the user to specify up to three values that will halt the 3230 CPU when they are encountered as a location counter value or a memory reference address value. A menu is displayed which allows setting of values and marking (with an asterisk) of their use as location value, memory reference, or both (See figure 2). Field-to-field movement is accomplished with the arrow keys. Placing the cursor on "go" in the menu initiates execution. ALT-F5 with the menu displayed exits the address setting mode without establishing any addresses and removes the menu. An ALT-F5 while running until a stop terminates the mode.

ALT-F6: Step Until Address Toggle

ALT-F6 toggles the "step-until" option for the 3230 CPU. This option allows the user to specify up to three values that will halt the 3230 CPU when they are encountered as a location counter value or a memory reference address value. A menu is displayed which allows setting of values and marking (with an asterisk) of their use as location value, memory reference, or both (See figure 2). Field-to-field movement is accomplished with the arrow keys. Placing the cursor on "go" in the menu initiates execution. ALT-F6 with the menu displayed exits the address setting mode without establishing any addresses and removes the menu. An ALT-F6 while running until a stop terminates the mode.

ALT-F7: Write Interpreter to Disk

ALT-F7 permits the user to write (edited) interpreter to disk. The user may specify a new file name or use the file name from which the original MetaMicro image was read. Files are written with the extension .PCH.

ALT-F8: Unused

Unused.

ALT-F9: Interpreter Load File

ALT-F9 makes it possible to specify a new interpreter file for the MetaMicro. The user is prompted for the file name and the MetaMicro memory image is read for subsequent loading into the MetaMicro. There follows a set of questions to complete interpreter loading. (See "Program Operation" for a full description.)

ALT F10 - DOS SHELL

Alt F10 invokes a second copy of the command interpreter (usually command.com) and allows you to run any program or batch job that can normally be run from the command line. To return to MMX you type EXIT (CR) at the prompt.

The command processor used is determined by the COMSPEC variable in the environment. When DOS is started it sets this variable to command.com.To use another command processor instead of command.com set COMSPEC correctly (see your DOS manual for directions for using the SET command).

The program on the metamicro will continue to run uninterrupted as long as neither screen nor disk I/O is requested. If I/O is needed the metamicro waits until control is returned to MMX to continue.

There are some precautions that should be taken while you are in a shell process. Never alter or delete any file that is being used by the metamicro, the results are unpredictable. Run only proven programs in the shell. If a program hangs the machine, work in progress in the MMX will probably be lost. Do not run another copy of MMX as it will not be initialized correctly.

SHIFT-F1: Debug Mode Toggle

SHIFT-F1 toggles the "debug mode". This mode causes the PC/MetaMicro I/O interface logic to display status information in the MetaMicro output portion (upper window) of the screen as they are executed. Status information about each transfer between the MetaMicro and the development system is displayed in the form shown in Figure 5 below. Debug mode output is distinguished by 'DEBUG:' at the beginning of each output

line. The "D" indicator at the top of the screen shows the on/off status of this switch.DEBUG: command = 9, unit = 10, flags = 0, ucb = -1

```
DEBUG: ctrlword = 31
DEBUG: DATA
DEBUG: 128 bytes read
DEBUG: 59 bytes returned
DEBUG: EOM
DEBUG: command = 9, unit = 10, flags = 0, ucb = 3
DEBUG: ctrlword = 31
DEBUG: DATA
DEBUG: STAT
DEBUG: error # 41
DEBUG: 0 bytes returned
DEBUG: EOM
DEBUG: command = 2, unit = 10, flags = 0, ucb = 3
DEBUG: ctrlword = 31
DEBUG: ACK
DEBUG: command = 4, unit = 6, flags = 0, ucb = -1
DEBUG: message # 41
DEBUG: crtlword = 31
41: "BEGIN TEST" PAUSE {}:
DEBUG: ACK
DEBUG: command = 18, unit = 0, flags = 0, ucb = 0
DEBUG: ctrlword = 31
DEBUG: ACK
DEBUG: command = 4, unit = 6, flags = 0, ucb = 3
DEBUG: message # 11
DEBUG: crtlword = 31
11: End of rule execution.
DEBUG: ACK
DEBUG: crtlword = 31
DEBUG: 2 bytes written
DEBUG: ACK
DEBUG: command = 2, unit = 6, flags = 0, ucb = 3
DEBUG: crtlword = 31
DEBUG: ACK
```

FIGURE 5 - Sample Debug Mode Output

SHIFT-F2: Register Display Toggle

SHIFT-F2 toggles the register display in the upper window of the screen. Interpreter only. Note that AR and BR are not user accessible registers. The interpreter knows them as inputs to the ALU.

SHIFT-F3: Stack Display Toggle

SHIFT-F3 toggles the stack display in the upper window of the screen. Interpreter only.

SHIFT-F4: External Bus Display Toggle

SHIFT-F4 toggles the external bus display in the upper window of the screen. Interpreter only.

SHIFT-F6: DUMP Memory, Data memory, Registers, Stacks

SHIFT-F6 prompts for requests to display the memory, data memory, registers, or stack. They are displayed in the upper window. For memory dumps, prompts request the starting location and the number of words to dump. ESC interrupts a long dump.

SHIFT-F8: Change Screen Display Mode

SHIFT-F8 cycles the screen display through three display modes: 1) full screen for the upper I/O window, 2) full screen for the lower instruction display window, and 3) half I/O window, half instruction window. The default is half-and-half.

SHIFT-F9: Toggle Programming Rule Checker

SHIFT-F9 toggles the rule checker in the interpreter and when the executive is running the 3230 CPU in step mode. This mode, which is normally on, checks for violations of progamming rules imposed by hardware timing, considerations. The user is told of programming constructs which do not allow sufficient time for register contents to be available before use and other rules discussed in section 3.

SHIFT-F10: SET

This function key allows the setting of several executive modes of operation. A prompt requests a single letter input:

I - Interrupt Register contents for interpreter
R - Rule checking mode
D - Disassembly mode
B - Bell mode

Interrrupt Register prompts for the value to be placed in the Interrupt Register and is only available in the interpreter.

Rule checking controls whether programming rule checking is done during stepping and stepping-until modes. In stepping mode errors are displayed in the lower screen window. In stepping-until mode the error is displayed in the upper window where it can be logged. Default is on.

Disassembly mode controls the instruction display: numeric or symbolic. The default is symbolic.

Bell mode lets you turn the error bell on or off. Default is on.

CTRL-F1: RUN - Start Interpreter

CTRL-F2: HALT - Stop Interpreter

CTRL-F3: Attention - Interrupt Interpreter

The three keys CTRL-F1 through F3 send status signals to the interpreter via setting status values and rasing a flag. Meanings of these unsolicited signals are determined by the interpreter for a particular high-level language. The meanings above are used by FORTRAN and other IMS products.

CTRL-F9: Assign I/O Units

This function allows the assignment and display of the correspondence between I/O unit number and DOS filename. Existing assignments are displayed, changes or new assignments can be made. New and changed assignments are filed permanently on prompted request.

At the prompt for a unit assignment number entering 'tab' displays all the current files assigned to units. Entering a unit number shows the current assignment which may be replaced with any valid DOS file name or accepted as is. The assignment is not checked until the file is opened.

CTRL-F10: Exit Development System

CTRL-F10 terminates execution of the PC/MetaMicro Development System, returning control to DOS. The user is prompted for a final yes/no confirmation for exit.

FUNCTION KEY SUMMARY

Key		Key	
1	Output Display Scroll-Up	2	Mic. Instruc. Scroll-Up
3	Output Down	4	Mic. Instruc. Down
5	Interpreter Run/Stop	6	Interpreter Single Step
7	Clear	8	Set Location
9	Load Interpreter	10	Display/Edit Interpreter

ALT		ALT	
1	PC I/O Printer Log ON/OFF	2	Interpreter Printer Log ON/OFF
2	Print PC I/O Last n Lines	4	Print Interpreter Last n Lines
5	Run Until Data or Instruction	6	Step Until Data or Instruction
7	Write Interpreter	8	----
9	Supply Interpreter File Name	10	Second DOS

SHIFT		SHIFT	
1	DEBUG I/O	2	* Register Display
3	* Stack Display	4	* External Bus Display
5	----	6	DUMP
7	----	8	Screen Toggle
9	Rule Check	10	SET

CONTROL		CONTROL	
1	RUN	2	HALT
3	Attention	4	----
5	----	6	----
7	----	8	----
9	Assign I/O Unit	10	EXIT

* interpreter only

PROGRAM OPERATION

To execute an already prepared MetaMicro program the following steps are required.

1. Load MMX by typing MMX at the DOS prompt or if already in MMX, use ALT-F9 to supply interpreter file name.

2. At the request for interpreter file name supply the full name of the file. A file extension of .MMC will bes assumed.

3. At the request for the file assignment file, the default, if any, will be displayed.

   a) CR will use the default
   b) a named file plus CR will use that file (.ASN will be assigned if no extension is given.)
   c) blank CR or ESC means no assign file

4. The next prompt provides for selective interpreter loading. There are three possibilities:

   a) CR to load the full interpreter. This is the normal case.
   b) P for partial loading. You will be prompted for beginning and ending addresses.
   c) N or ESC to not load interpreter.

5. The next prompt asks if you are ready to run. CR or Y begins execution, any other key takes you to the MMX "waiting" state for function key commands.

III MICROPROGRAMMING

The 3230 CPU processor includes twenty-four hardware operations that are combined into a two instruction per word format. There are eight op codes in the left hand side (LHS) and sixteen op codes in the right hand side (RHS). A typical composite instruction consists of:

LHS Portion: an arithmetic, logic, or shift operation between two operands with the result assigned to a third operand; and

RHS Portion: an external bus (unidirectional or bidirectional) operation, a conditional/ unconditional transfer/skip operation, a subroutine link/return operation, an interrupt handling operation, or a memory indexing operation.

A typical composite instruction, including a shift and a bidirectional bus operation, is the following:

LHS	RHS
R5 := R0 SLR 4	BUS EMIT MEMWORD REC MEMSTATUS

This composite instruction performs a left rotational shift of general register R0 by four bits, stores the result in general register R5 as well as the X (result) register, then emits the contents of the X register to the external bus address "MEMWORD" (an assembley-parameter) and initiates the receipt of the status information from bus address "MEMSTATUS". The bus operations are executed as the next instruction executions take place. The result of the bus REC operation ultimately appears in the external input register (EIR). See 2.6.8 for details.

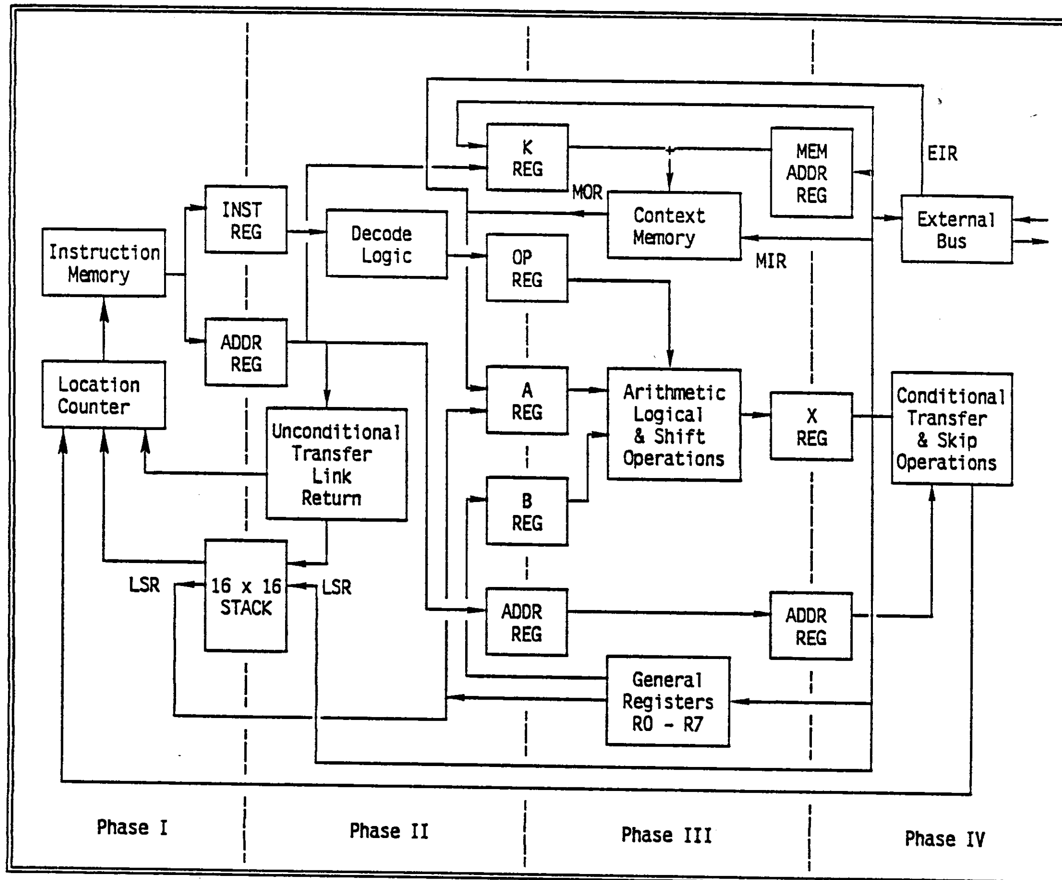


FIGURE 6 - Four-Phase Pipeline Operation

INSTRUCTION PIPELINE

To achieve the high instruction throughput rate of 30 to 50 million instructions per second, the 3230 CPU is implemented as a "pipelined" computer with the following four phases. (Phases are shown in Figure 6).

1. Composite instruction fetch

2. Composite instruction decode and operand fetch, execution of RHS unconditional transfer, link, return, interrupt conditional link or memory indexing operations

3. Arithmetic/Logic/Shift execution

4. Conditional transfer/skip or external bus I/O initiation

Complete execution of a composite instruction requires that it pass through the four phases listed above. Each phase requires that one clock cycle and four composite instruction words are in the process of execution (one in each phase) at any point in time. A composite instruction is completed each clock cycle. Most composite instruction words are the equivalent of two complete instructions of a typical CISC or RISC architecture.

In Figure 6 the register positioning on a phase boundary indicates that the information is provided for that register by the preceding phase and is available to the succeeding phrase.

LEFT HAND SIDE FORMAT

The LHS of the composite instruction usually has the form:

<T> := <A> <u>op</u> <B>

where <T> is a register, := denotes "assign", <A> is a register or the literal zero, and <B> is a register or a literal. The <u>op</u> token is one of eight arithmetic, logic, or shift operators.

The LHS instructions in assembly format are described briefly below. Arithmetic and logical operation apply to either short or long operands.

Operators	Function
+	Add
-	Subtract
- +	Reverse Subtract
XOR	Exclusive OR
OR	Logical OR
AND	Logical AND
SLR, SLL	Left Shift: rotational and logical
SRR. SRL	Right Shift: rotational and logical

Examples: (leading period denotes start of comment)

```
R5  := R0 SLL 6              .shift R0 left-logical 6 bits,
                             .assign to R5 and X
R0  := X XOR R5              .XOR X with R5, assign to R0
X   := R3 + KR               .add KR to R3, assign to X
MIR := R5 AND X'7FFFF'       .R5 and hex literal, store in
                             data memory
```

The registers used in these instructions are described briefly below:

Register(s)	Name	Size (bits)
R0 ... R7	General Registers	32
X	Result Register	32
MAR	Memory Address Register	16
MIR	Memory Input Register	32
MOR	Memory Output Register	32
KR	K Register	16
LSR	Link Stack Register File	16 x 16
IR	Interrupt Register	8
IMR	Interrupt Mask Register	8
EIR	External Input Register	32
DATA	Next instruction used as 32-bit literal	32

RIGHT HAND SIDE FORMAT

The RHS of the composite instruction has the form:

<code> <phrase>

where <code> is one of sixteen operations, including external bus read/write/control, subroutine link/return, conditional interrupt handling, conditional and unconditional transfers and skips and memory indexing. The <phrase> is a structure of one or more operands and keywords.

The RHS instructions are summarized below:

Operator	*Function*
TRA	*Transfer unconditionally*
TRA ... ON	*Transfer conditionally on Overflow or not overflow*
	X reg low bit 0 or 1
	X reg high bit 0 or 1
	X reg zero or non-zero
SKIP	*Skip conditionally*
LINK	*Subroutine link*
LINK C	*Link conditional on interrupt*
RETURN	*Subroutine link*
LDK	*Load K register*
BUS FROM...TO	*Initiate bus source to des tination transfer*
BUS EMIT Recieve	*X register to bus destination and bus source to EIR*

Examples:

```
... LINK PROC1 .call microsubroutine PROC1, push return
               address on Link Stack Register (LSR) File

... SKIP 3 IF STATUS HAS NLB  .skip 3 instructions on
                              condition "not low bit"

... BUS EMIT MEMC   .emit X register to bus
                    .address MEMC

... TRA LOOP3 ON Z  .transfer to LOOP3 if X register is
                    zero
```

... RETURN DELTA *.subroutine return to top LSR value plus DELTA value, pop LSRThe LHS and the RHS are combined into an instruction that executes in a single 3230 CPU clock cycle. A clock cycle delay can be introduced by the use of the no-operation, NOP. Also the NOP command is used for a null LHS where only an RHS is desired in a composite instruction. NOP is an assembler pseudo op that generates the LHS institution X:= 0 SLR 1.*

Composite Examples:

X:=R5 SLR 6 TRA PROC3 ON HB *.shift, assign,*
.transfer if negative

MIR:=X+2 SKIP 2 IF STATUS HAS NHB *.add, assign,*
.skip conditional

X:=R2+R3 BUS EMIT DES1 REC DES4 *.add, assign,*
.bus emit
.bus receive

R3:=X-R4 TRA L30 ON Z *.subtract, assign,*
.transfer on zero

NOP TRA LOC16 *.unconditional transfer with LHS negated*

R4:=X SRR 4 RETURN 6 *.shift, assign, subroutine return plus 6*

OFF-CHIP INTERFACES

Interfaces between the 3230 CPU processor and other subsystems are through the special registers and the external bus operations. The data memory is accessed using the MIR, MAR, MOR, and the K register in LHS operations. The user memory and the PC-channel are accessed via the RHS external bus operations. The BUS EMIT operation sends the contents of the X register to a specified bus destination. The BUS REC operation receives an input into the external input register (EIR) from a specified bus source.

1.0 SPECIAL-PURPOSE REGISTERS

In addition to the eight general-purpose 32-bit-wide registers provided for the temporary storage of intermediate results, a set of special-purpose registers provide program interfaces to special hardware logic. The term "source" is used to designate registers used as sources of input operands <A> and <B>; the term "result" is used to designate registers used as the object of the assign operation. Certain registers may be used both as source and result.

1.1 X REGISTER

All instructions, including NOP, produce a 32-bit-wide result as a consequence of an Arithmetic/Logical/Shift operation. This result can be directed to one of the general-purpose registers, to one of the special-purpose registers, or neither. However, the result will always reside in the X register as well as in any general- or special-purpose register specified. In the symbolic assembly language, the X register is often specified as the result register and classed as a special-purpose register. When so specified, it has the meaning "the X register and no other register".

1.2 MEMORY ADDRESS REGISTER (MAR)

The MAR is 16 bits wide and is used to hold an address which, summed together with the contents of the "K Register" (see 1.5 below), selects a memory location in the Data memory memory.

1.3 MEMORY INPUT REGISTER (MIR)

The MIR can be used only as a result register, not as an operand source. A result directed to the MIR will be stored in the memory location within the data memory memory as specified by the sum of the MAR and the K Register.

1.4 MEMORY OUTPUT REGISTER (MOR)

The MOR can be used only as an operand source, not as a result register. It specifies that the contents of the memory location of the Data memory memory as addressed by the sum of the K Register and the MAR is selected as an operand source.

1.5 K REGISTER

The K register is a multi-function, 16-bit-wide register that exhibits many of the characteristics of an index register in a conventional single-address computer. As specified in 1.3 above, the K register, in summation with the MAR, forms the memory address used to reference the data memory memory. It can also be used like a general register. See 2.6.2 for more details.

1.6 LINK STACK REGISTER (LSR)

The LSR is a 16-bit-wide, 16 element last-in-first-out-(LIFO) register stack whose primary function is to implement the subroutine "LINK" and "RETURN" instructions of the 3230 CPU. However, it can also be used as a source or result register. Its use as a source is equivalent to a POP, while its use as a result is equivalent to a PUSH. When conflicts arise (as when the LSR appears in the "T" field preceding a Link or Return instruction), the use of the LSR for "LINK" and "RETURN" takes precedence over the use as a source or result register. Another way to view LSR is as a window onto the 16 element stack. PUSH moves the window up and stores; POP fetches and moves the window down. The stack is circularly connected: 16 POPs leave the stack unchanged. There are no fill or empty stack conditions.

1.7 DATA REGISTER (DR)

The DR can be used only as an operand source, not as a result register. The DR is not a specific hardware register; when specified it means that the next instruction is to be considered a 32-bit data value. The assembler directive DC is often used to define the value for the data registers.

1.8 INTERRUPT REGISTER (IR)

The IR can be specified both as a source and result register. It is 8 bits wide and represents 8 boolean flags that can be set (=1) by events and conditions external to the scope of the 3230 CPU. The bits can be reset (=0) by the 3230 CPU. When used as a source register, the IR represents an 8-bit-wide value, each bit of which is the state or condition of an independent variable in a set of boolean variables. When used as a result register, writing a value into the register will reset those boolean variables for which a corresponding "one" bit exists in the value.The IR and the IMR work in conjunction with the Link Conditional instruction (See 2.64 and 3.2)

Assignments of the bits in the interrupt register are :

Bit	Set Condition
0	External bus parity error
1	I/O Timeout, IOTMO
2	Data memory parity error
3-7	Unused

1.9 INTERRUPT MASK REGISTER (IMR)

The IMR can be used only as a result register, not as an operand source. It is 8 bits wide and represents a mask for the 8 boolean flags of the Interrupt Register (IR). The IMR and the IR operate in conjunction with the Link Conditional instruction (see 2.6.4 and 3.2).

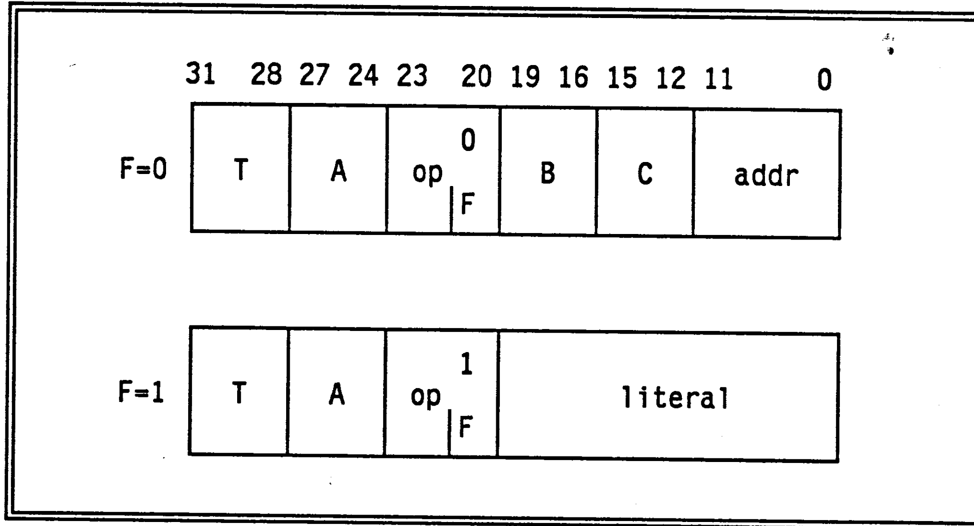
1.10 EXTERNAL INPUT REGISTER (EIR)

The EIR is the input termination of the MetaMicro External Bus (EB). This register can be used as a source operand only and is loaded by action on the EB. As the EB operates asynchronously from the operation of the 3230 CPU, special consideration must be given to its use. When an EB operation that results in data input to the 3230 CPU (the EIR) is initiated, the EIR contents are undefined until the bus operation is completed. If the EIR is referenced by the 3230 CPU before the bus operation is complete, the 3230 CPU will stop and wait for completion, thus synchronizing the 3230 CPU and the EB. However, if synchronization does not take place within 16 3230 CPU clock cycles,

operation will proceed unconditionally and a positive pulse will be generated on the external pin labeled IOTMO (input/output timeout). This signal sets bit 1 of the Interrupt Register, IR. The value of the EIR under forced continuation circumstances is the previous EIR value.

2.0 INSTRUCTION FORMAT

The 3230 CPU employs two forms of instruction encoding as shown below, designated composite (F=0) and LHS-only (F=1).



In assembler format, instructions consist of two statements:

LHS: arithmetic, logical, or shift statement (or a NOP), followed by

RHS: optional statement, providing a second instruction of up to 16 alternative operations, including external bus instructions, subroutine link/return (both conditional and unconditional), conditional skips, secondary loading of the K register, and conditional/unconditional transfers.

<instruction> :: = <LHS_stmt> [<RHS_stmt>]

For LHS Arithmetic/Logical/Shift operations the 3230 CPU employs a three-address instruction format that can be represented symbolically as a simple assignment statement of the form:

<T>:=<A>op<B>

wherein a register specified by the "T" field takes on the value of some binary operation (op) performed on the contents of the register specified by the "A" field and the contents of the register specified by the "B" field. In the case of literal or immediate data

representation of magnitude < 8, the "B" field may represent the data directly.

In assembler format, these statements have the following general form:

```
<LHS_stmt> ::=
<T_field>:= [-] <A_field> <op> <B_field> | NOP
```

Examples:

```
X   := X SLL 4
IMR := R0 XOR R4
NOP
```

2.1 "T" (TARGET OPERAND) FIELD

The "T" field (bits 31-28) is four bits wide and references the registers as shown in the table below.

In assembler format, the "T" field is defined as follows:

```
<T_field> ::=
<gen_reg> | MAR | MIR | X | KR | LSR | IR |IMR
```

where

```
<gen_reg>::=R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7
```

The "T" field assembler mnemonics and equivalences are given in the following table:

T FIELD			
HEX	BINARY	MNEMONIC	MEANING
0	0000	(not used)	---
1	0001	X	X REGISTER
2	0010	MAR	MEMORY ADDRESS REGISTER
3	0011	MIR	MEMORY INPUT REGISTER
4	0100	KR	K REGISTER
5	0101	LSR	LINK STACK REGISTER (push
6	0110	IR	INTERRUPT REGISTER
7	0111	IMR	INTERRUPT MASK REGISTER
8	1000	R0	GENERAL REGISTER 0
9	1001	R1	" " 1
A	1010	R2	" " 2
B	1011	R3	" " 3
C	1100	R4	" " 4
D	1101	R5	" " 5
E	1110	R6	" " 6
F	1111	R7	" " 7

2.2 "A" (PRIMARY INPUT OPERAND) FIELD

The "A" field (bits 27-24) is four bits wide and references the registers in the table below.

In assembler format, the "A" field is defined as follows:

<A_field> ::=
<gen_reg> | 0 | X | EIR | MOR | KR | LSR | IR | DATA

The "A" field assembler mnemonics and equivalences are given in the following table:

A FIELD			
HEX	BINARY	MNEMONIC	MEANING
0	0000	0	THE LITERAL ZERO
1	0001	X	X REGISTER
2	0010	EIR	EXTERNAL INPUT REGISTER
3	0011	MOR	MEMORY OUTPUT REGISTER
4	0100	KR	K REGISTER
5	0101	LSR	LINK STACK REGISTER (POP)
6	0110	IR	INTERRUPT REGISTER
7	0111	DATA	DATA REGISTER
8	1000	R0	GENERAL REGISTER 0
9	1001	R1	" " 1
A	1010	R2	" " 2
B	1011	R3	" " 3
C	1100	R4	" " 4
D	1101	R5	" " 5
E	1110	R6	" " 6
F	1111	R7	" " 7

2.3 OPERATOR FIELD

The "op" field (bits 23-20) is four bits wide[2] and specifies the Arithmetic/Logical/Shift operation that is to be performed on the data sources specified by the "A" and "B" fields. The encoding of the "op" field is Shown in the table below.

In assembler format, the "op" field takes the form:

<op>:: = <A_op> | <L_op> | <S_op> | <No_op>

---

[2] Bit 20 (the "F" field) is used as a suboperation qualifier and format modifier to distinguish between rotational and logical shifts and to specify whether the "B" field is "short" (register operand or 3-bit literal) or "long" (a 20-bit literal in the "C" and address fields).

The arithmetic operators include addition, subtraction, and reverse subtraction (indicated by using a + with a - before the A_field):

<A_op>::= + | -

The logical operators are OR, AND, and XOR:

<L_op>:: = OR | AND | XOR

The shift operators are logical shift left or right and rotational shift left or right.

<S_op>::=SLL | SRL | SLR | SRR

The <No_op> (empty operation) statement,

<No_op> ::= NOP

will generate the code:

X := 0 SLR 1

The "op" field assembler mnemonics and equivalences are given in the following table:

OP FIELD			
HEX	BINARY	MNEMONIC	MEANING
0	0000	SLR	SHIFT LEFT ROTATIONAL
1	0001	SLL	SHIFT LEFT LOGICAL
2	0010	- +	REVERSE SUBTRACT
3	0011	- +	REVERSE SUB. 20-BIT LITERAL
4	0100	-	SUBTRACT
5	0101	-	SUBTRACT 20-BIT LITERAL
6	0110	+	ADD
7	0111	+	ADD 20-BIT LITERAL
8	1000	XOR	EXCLUSIVE OR
9	1001	XOR	EXCLUSIVE OR 20-BIT LITERAL
A	1010	OR	LOGICAL OR
B	1011	OR	LOGICAL OR 20-BIT LITERAL
C	1100	AND	LOGICAL AND
D	1101	AND	LOGICAL AND 20-BIT LITERAL
E	1110	SRR	SHIFT RIGHT ROTATIONAL
F	1111	SRL	SHIFT RIGHT LOGICAL

Examples:

```
NOP
  X := R0 + R2
IMR := -X + R0
 R0 := 0-1
 R7 := R5 AND 1
  X := R1 OR X'7FFF'
 R5 := MOR AND 1
  X := MOR XOR NAME_T
  X := R0 SLL 6
 SR := R5 SRL 6
  X := R7 SRR 2
```

2.4 "B" (SECONDARY INPUT OPERAND) FIELD

The "B" field (bits 19-16) is four bits wide in its short form (F bit = 0). For short form operations involving arithmetic/logical manipulation (op=2,4,6,8,A,C), the "B" field references the registers and or literals as shown in the table below:

In assembler format, the short form "B" field may be either a general register or an expression representing the literals (0-7).

```
<B_field>::=<gen_reg> | <expr>
```

The assembler mnemonic equivalences for arithmetic and logical operations are given in the following table:

B FIELD - Arithmetic & Logical			
HEX	BINARY	MNEMONIC	MEANING
0	0000	EXPRESSION	3 BIT-LITERAL = 0
1	0001	"	" = 1
2	0010	"	" = 2
3	0011	"	" = 3
4	0100	"	" = 4
5	0101	"	" = 5
6	0110	"	" = 6
7	0111	"	" = 7
8	1000	R0	GENERAL REGISTER 0
9	1001	R1	" " 1
A	1010	R2	" " 2
B	1011	R3	" " 3
C	1100	R4	" " 4
D	1101	R5	" " 5
E	1110	R6	" " 6
F	1111	R7	" " 7

For operations involving shifting (op=0,2,E,F), the "B" field specifies the shift values as listed in the table below.

For shift operations, the "B" field must be an expression whose value is between 1 and 8, inclusive, which are converted by the assembler to the values 0-7.

The assembler mnemonic equivalences are given in the following table:

B FIELD - Shifts			
HEX	BINARY	MNEMONIC	MEANING
0	0000	EXPRESSION	SHIFT 1
1	0001	"	" 2
2	0010	"	" 3
3	0011	"	" 4
4	0100	"	" 5
5	0101	"	" 6
6	0110	"	" 7
7	0111	"	" 8

2.5 OVERFLOW IN ARITHMETIC/LOGICAL/SHIFT INSTRUCTIONS

The arithmetic operations, Add, Subtract, and Reverse Subtract may generate an arithmetic overflow if the result generated by the operation extends beyond bit 31 into a bit known as the overflow bit. The overflow bit is also used for left, right, logical, and rotational shifts.

Caution on subtract overflow: Subtraction is accomplished by forming the 1's complement of the subtrahend and adding it, plus 1, to the minuend. If this addition overflows, the overflow bit is set. The result is that the overflow bit is set if underflow does not occur (6-5 overflows, 6-0 overflows, 5-6 does not overflow).

For logical shifts, data leaving either the high order bit (bit 31) or the low-order bit (bit 0) for left and right shifts, respectively, reside in the overflow bit position. Thus, following a right or left logical shift, if the last bit to be shifted out of the result was a one, then the overflow bit will be a one. Conversely, if the last bit to be shifted out of the result was zero, then the overflow bit will be a zero.

For rotational shifts, bits shifted out of bit position 31 reenter the result in bit position zero (left rotational) or bits shifted out of bit position 0 reenter the result in bit position 31 (right rotational).In

either case, if a one bit traverses from position 31 to 0 or from 0 to 31, then the overflow bit will be set. If no one bits traverse through, the overflow bit will not be set. For example, during a right rotational shift of eight bits, if any of the 8 bits of the input in positions 7 through 0 were a one, then the overflow bit would be set in the result. Conversely, if all of the 8 bits of the input in positions 7 through 0 were zeros, then the overflow bit would not be set in the result.

2.6 "C" (RIGHT HAND SIDE) FIELD

When the "B" field has short form, the "C" and Address fields constitute the right-hand-side RHS instruction in which the "C" field is an operation code and the Address field is a memory address reference or a literal field. The encoding of the "C" field is shown in the table below.

An assembler "RHS_statement" may be included with a shift statement, with any arithmetic or logic statement not using a 20-bit literal, or with a NOP. If an RHS_statement is allowed but not used, then the assembler will generate code for a skip statement that has no effect.

<RHS_statment> ::=

<link_return_stmt> | <load_k_stmt> | <transfer_stmt>|
<bus_stmt> | <ski_stmt>

Note: When a shift operator is used, the B_field must be an expression whose value is between one and eight, inclusive. If an arithmetic or logic operator is used and the B_field is an expression with value between 0 and 7, inclusive, then a 3-bit literal will generated. If the B_field is an expression with value outside this range, then a 20-bit literal will be generated and any RHS_statement will be ignored. In this case, the assembler will generate an appropriate error message.

The assembler mnemonic equivalences for RHS_statements are given in the following table:

C FIELD			
HEX	BINARY	MNEMONIC	MEANING
0	0000	SKIP	SKIP CONDITIONAL
1	0001	LDK	LOAD K REGISTER
2	0010	BUS	BUS (FROM/TO)
3	0011	BUS	BUS (EMIT/REC/EMIT-REC)
4	0100	LINK	LINK TO SUBROUTINE
5	0101	LINKC	LINK COND. ON INTERRUPT
6	0110	RETURN	SUBROUTINE RETURN
7	0111	TRA	TRANSFER UNCONDITIONAL
8	1000	TRA	CONDITIONAL TRANSFER ON NOV
9	1001	TRA	" " ON OV
A	1010	TRA	" " ON NLB
B	1011	TRA	" " ON LB
C	1100	TRA	" " ON Z
D	1101	TRA	" " ON NZ
E	1110	TRA	" " ON NHB
F	1111	TRA	" " ON HB

2.6.1 CONDITIONAL TRANSFERS, TRA ([C]=8...F)

Conditional transfer instructions examine the result of the arithmetic/logical/shift portion of the instruction (<T>:=<A><u>op</u><B>) and cause control to be transferred to the memory address in the Instruction memory specified by the Address field if the associated condition is true. Because the 3230 CPU is a "pipeline" machine, the consequences of this action are more complex than that of a "non-pipeline" machine.

As stated earlier, the conditional transfer portion of the instruction is executed in the fourth phase of the pipeline. Therefore, the three instructions immediately following the conditional transfer instruction are already in the pipeline and partially executed.

Specifically, as Figure 6 shows, the first instruction is in the arithmetic/logical/shift circuitry (Phase III); the second is undergoing decoding (Phase II) and right-hand-side execution; and the third is being fetched (Phase I). Since the transfer condition is not

determined until Phase IV, an additional effect of the conditional transfer instruction is to flush these instructions if the condition is true and the transfer is taken.An exception is the next instruction, which has already completed Phase II and is in Phase III. The Left hand side has not completed execution, but the A and B registers have been loaded (possibly popping the stack). If it includes in its RHS, a transfer, link, return, or load K operation (which execute in Phase II), it will already have been executed, and the location counter changed. This location counter change will be superseded, since the conditional transfer will reset the location counter to the desired transfer location when the transfer condition is met. The load K instruction will be executed, altering the K register. Also, the push or pop by the link or return RHS instruction will take place. Three cycles are lost in flushing the pipeline following the true execution of a conditional transfer.

In assembler format, the conditional transfer instructions have the following form:

```
<transfer_stmt>::= TRA <expr> ON <tra_cond>
<tra_cond>::= Z | NZ | OV | NOV | LB | NLB | HB | NHB
```

Examples:

```
... TRA @-3 ON NZ  (@ ::= this instruction address)
... TRA ERROR_16 ON OV
```

Composite Examples:

```
R7 := MOR+0 TRA INTERPT_1B ON Z
X  := R2 SLL 1 TRA NOMORE ON Z
```

The transfer conditions have the following meanings:

```
Z   ::= ZERO
NZ  ::= NON-ZERO
OV  ::= OVERFLOW
NOV ::= NO OVERFLOW
LB  ::= LOW BIT = 1
NLB ::= LOW BIT = 0
HB  ::= HIGH BIT = 1
NHB ::= HIGH BIT = 0
```

2.6.2 LOAD K REGISTER, LDK ([C]=1)

The K register is a special-purpose register used in conjunction with the Memory Address Register (MAR) to reference the Data memory memory. The K register can be used as a conventional source or result operand in an instruction and can also be loaded directly with a 12- bit value (the address field) under control of the "C" field.The K register is 16 bits wide; consequently, when the K register is loaded under "C" field control, the upper four bits (bits 15-12) of the K register are cleared to zeros. The loading of the K register takes place at the end of Phase II of the pipeline (see 3.2). Consequently, the value loaded will appear in the register one instruction cycle after the instruction used to load it.

In assembler format, the LDK statement is used to load the K register with the value of the expression.

<load_k_stmt>:: = LDK <expr>

Examples:

```
... LDK X'FFF'
... LDK 536
... LDK (1-X)*Y
```

Composite Examples:

```
R1 := R2 SLL 8 LDK 3
MAR := MIR-1 LDK HEAP_OFFSET-2
```

2.6.3 UNCONDITIONAL TRANSFER, TRA ([C]=7)

As the name implies, unconditional transfer always causes a transfer of control in the Instruction memory program. Since there are no dependencies on various machine conditions for the transfer to take place, the unconditional transfer is implemented within the pipeline at the earliest possible position which is in Phase II. When the transfer is recognized, the subsequent instruction in Phase I has already been fetched from Instruction memory and executes normally. Consequently no clock cycle is lost. This facility, while somewhat unique, can be put to very good use in the programming for the 3230 CPU.

In assembler format, the unconditional transfer statement has the following form:

<unconditional_transfer> ::= TRA <expr>

Examples:

```
... TRA CHANNEL3
... TRA @-3
```

Composite Examples:

```
R2 :=MOR+0 TRA INTERPT_1
    NOP   TRA INTRUPT_1K
```

2.6.4 LINK INSTRUCTIONS, LINK ([C]=4)

The Link instruction ([C]=4) not only produces an unconditional transfer, but also causes the value of the address plus two of the link instruction to be "pushed" onto a sixteen-element LIFO stack register file (the Link-Stack Register). This saves the address of the Link instruction for subroutine return.

In assembler format, the link instruction has the following form:

<link_stmt> ::= LINK <expr>

Examples:

```
... LINK SUBROUT1
... LINK @+6
```

Composite Examples:

```
R1 := MOR-2 LINK EXT_BLK
NOP  LINK INTERPT_15B
```

2.6.5 LINK CONDITIONAL, LINKC ([C]=5)

The Link Conditional instruction ([C]=5) is a specialized instruction that is dependent upon interrupt conditions. It is similar in function to the Link instruction but is executed only if the special interrupt test conditions are true.

The test consists of examining the individual bits of the Interrupt Register (IR) for which a corresponding bit position in the Interrupt Mask Register (IMR) is a "1". The test is false if there is not at least one matching bit pair; no transfer or Link Stack "push" takes place. If one or more matching bit pairs exist, then a value equal to the position of the highest-priority bit pair is formed. Position "0" is the highest-priority bit position and position "7" is the lowest-priority bit position. The value formed is doubled and added to the address field of the instruction to form a transfer address. In addition, the bit position within the IR corresponding to the highest priority bit position is reset or cleared.

In summary, a subroutine linkage is conditionally made to an address which is the value of the Address field plus twice the value of the highest priority masked bit position in the IR. The bit position within the IR is reset.

In assembler format, the link conditional statement has the following form:

<link_conditional_stmt> ::= LINKC <expr>

Examples:

```
...    LINKC INTRUPT_A6
...    LINKC @+4
```

Composite Examples:

```
IMR := EIR AND R0 LINKC INP_INTR
NOP  LINKC RESET_FLAG
```

2.6.6 RETURN INSTRUCTION, RETURN ([C]=6)

The Return instruction ([C]=6) causes an unconditional transfer to the address formed by summing the Address portion of the Return instruction and the "top" element of the Link-Stack Register File. The top stack element is "popped" (deleted).

In assembly format, the Return instruction has the form:

<return_stmt> ::= RETURN [<expr>]

Examples:

... *RETURN 2*
... *RETURN OFFSET4*

Composite Examples:

R0 := 0-1 RETURN INTERPT_3A
NOP RETURN 0

2.6.7 SKIP CONDITIONAL INSTRUCTION, SKIP ([C]=0)

A Skip Conditional instruction, like the Conditional Transfer instruction, tests the result of the arithmetic/logical/shift portion of the instruction (<T>:=<A>op <B>). However, multiple and more complex tests can be performed and the result, if true, is to cause from one to four subsequent instructions to be skipped. For each instruction skipped, one clock cycle is lost. Because the test associated with the skip instruction is made in Phase IV of the pipeline, the instruction immediately following the skip instruction is already in Phase III when the test is made. Further, this instruction has already passed through Phase II and with regard to the RHS (transfer, link, return or load K) portion of the instruction, hasalready been executed. Therefore, while the assignment portion of the instruction which immediately follows a skip instruction can be negated, if the following instruction has a RHS portion, that portion of the instruction will execute normally. If a transfer, link, return, or load K is present in any subsequent instruction which is to be skipped, it will be skipped entirely (see examples below).

For skips of three or four instructions and where the instruction immediately following the skip instruction contains a transfer, link or return sub-function, the instructions skipped are (1) the two instructions immediately following the skip instruction and (2) one or two subsequent instructions beginning at the point of the transfer address.

The address field of the skip instruction does not contain a memory address; rather, it is used to specify the type of skip, the extent of the skip, and a mask which defines the conditions of the test. The type of skip is encoded into bits 9-8 of the address field as listed below.

Type = 0 If the Exclusive OR of the STATUS and the MASK are all zeros, then skip.

Type = 1 If the Logical AND of the STATUS and the MASK is anywhere non-zero, then skip.

Type = 2 If the Exclusive OR of the X Register (7-0) and the MASK are all zeros, then skip.

Type = 3 If the Logical AND of the X Register (7-0) and the MASK is anywhere non-zero, then skip.

The MASK is an eight-bit field specified by bits 7-0 of the address field. The STATUS is an eight-bit value formed by the hardware in Phase IV from the overflow bit and the X Register contents as follows.

Status Bit 0: Set if Overflow bit is not set
Status Bit 1: Set if Overflow bit is set
Status Bit 2: Set if X Register bit 0 is not set
Status Bit 3: Set if X Register bit 0 is set
Status Bit 4: Set if all X Register bits are zero
Status Bit 5: Set if any X Register bit is non-zero
Status Bit 6: Set if X Register bit 31 is not set
Status Bit 7: Set if X Register bit 31 is set

Thus, skip types 0 and 1 allow multiple status conditions to be tested in combination whereas skip types 2 and 3 allow the result of the T:= A op B to be tested in the low order (bits 7-0) eight bit positions.The extent of the skip, if taken, is encoded in bits 11-10 of the address field as listed below.

Extent = 0: Skip 1 instruction
Extent = 1: Skip 2 instructions
Extent = 2: Skip 3 instructions
Extent = 3: Skip 4 instructions

In assembly format, the skip statement has the form:

<skip_stmt> ::= SKIP <extent_expr>
[[IF] (STATUS | RESULT)
(IS | HAS) <mask_expr>]

This statement uses the keyword IS to represent XOR, since the purpose of the XOR is to test for bitwise equivalence of the STATUS or RESULT fields and the mask expression. HAS is used to represent AND, since the purpose of the test is to detect common bits in the STATUS or RESULT fields and the mask

expression. The assembler will generate an unconditional skip if the IF clause is not present.

Examples:

```
... SKIP 1 IF STATUS IS X'11'
... SKIP 4 IF RESULT HAS X'FC'
... SKIP 3
```

Composite Examples:

```
R0 := R4 SLL 4 SKIP 2 IF RESULT HAS LB
R6 := X+R4 SKIP 3 IF STATUS HAS OV
```

2.6.8 EXTERNAL BUS INSTRUCTIONS, BUS ([C]=2,3)

Bus From-To Instruction ([C]=2)

The external bus 'FROM-TO' Instruction (C=2) causes a subsystem on the external bus to act as a source (transmitter) and to place information on the 32 data lines of the external bus. A second subsystem activated by the command acts as a destination (receiver). The 3230 CPU thus acts as an initiator and monitor of the bus activity but does not itself participate as source or destination.

The address field of the instruction initiating the external bus operation is formatted to contain two six-bit addresses. Bits 11-6 contain the bus source subsystem address, and bits 5-0 contain the bus destination subsystem address. Once issued, the bus operation can proceed independently of the initiating 3230 CPU. However, the bus itself becomes active and cannot be used forsubsequent operations until the current activity is completed. If the initiating 3230 CPU should subsequently issue a second bus operation before the current operation has completed, the 3230 CPU will stop and wait for completion. Should the operation not complete within an additional 16 3230 CPU clock cycles, the 3230 CPU will unconditionally terminate it by removing the source and destination addresses from the external bus control logic. In this event, a positive pulse will be generated on the external pin labeled IOTMO (input/output timeout). This signal is used to set interrupt register bit 1.

In assembly format, the external bus FROM-TO statement has the following form:

<bus_from_to_stmt> ::= BUS FROM <source>
TO <destination>

where <source> and <destination> are expressions.

Examples:

```
...   BUS FROM DRAM TO FPINPUT
...   BUS FROM NODE1 TO NODE2
```

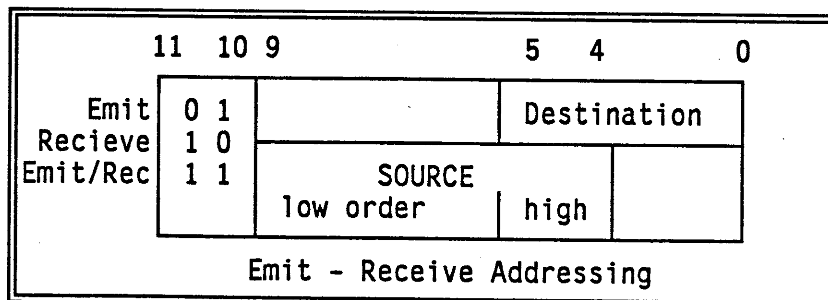
Composite Examples:

```
X := R0 SLL 4 BUS FROM FPOUTPUT TO DRAM
NOP   BUS FROM NODE2 TO NODE4
```

Bus Emit-Receive Instruction ([C]=3)

The external bus 'EMIT-RECEIVE' Instruction (C=3) operates like that described above except that the initiating 3230 CPU is always involved directly as a bus source, destination or both. Three modes of operation are possible, designated EMIT, RECEIVE and EMIT-RECEIVE. Bits 11-10 of the address field of the initiating instruction specifies one of the three modes of operation as follows;

EMIT	bit 11=0, bit 10=1
RECEIVE	bit 11=1, bit 10=0
EMIT-RECEIVE	bit 11=1, bit 10=1If both bits (11 and 10) are zero, the operation is a 'No-Operation' and initiates no bus activity.



Emit - Receive Addressing

For the EMIT (from 3230 CPU to subsystem) mode of operation, bits 5-0 of the address field contain a 6-bit bus destination subsystem address. The source is

the 3230 CPU itself, and the source data (32 bits) is the value of the X register produced by the initiating instruction.

For the RECEIVE (from subsystem to 3230 CPU) mode of operation, bits 5-4 and bits 9-6 of the address field contain a 6-bit bus source subsystem address. The destination is the 3230 CPU itself and the receiving register is the EIR.

For the EMIT-RECEIVE mode of operation, the two operations, EMIT and RECEIVE, are initiated in sequence. The RECEIVE operation is delayed until the completion of the EMIT operation has been detected by the 3230 CPU's External Bus logic. The EMIT and RECEIVE subsystem addresses are encoded as described for the individual bus operation modes. As bits 5-4 are common to both bus addresses, there are restrictions as to which subsystems can be involved in an EMIT-RECEIVE bus operation. Obviously, only subsystems which have bit values for bits 5-4 in common can be involved. Subsystem addresses are normally established by strappings or switch settings within the subsystems themselves and thus the restriction relative to bits 5-4 is relatively easy to accommodate.

For all three modes described above, if the initiating 3230 CPU should subsequently issue a second bus operation before the current operation has completed, the 3230 CPU will stop and wait for completion. Should the operation not complete within an additional 16 3230 CPU clock cycles, the 3230 CPU will unconditionally terminate it by removing the source and/or destination addresses from the bus control logic. In this event, a positive pulse will be generated on the external pin labeled IOTMO (input/output timeout). This signal sets interrupt register bit 1.Standard MAX 2 bus addresses (and included comments) and status values are given in the assembler definition files and listed in Appendix A, Section A3.

In assembler format, external bus EMIT_RECEIVE statements have the following form:

```
<bus_emit_rec_stmt> ::= BUS {<emit_stmt> | <rec_stmt>|
                             <emit_stmt> | <rec_stmt>}
     <emit_stmt>    ::= EMIT <destination_expr>
     <rec_stmt>     ::= REC <source_expr>
```

Examples:

```
...  BUS EMIT BIGMEM
...  BUS REC MEMUNIT1
...  BUS EMIT MEM1 REC MEM1_STATUS
```

Composite Examples:

```
X := R3+0 BUS EMIT MEM1_CNTL REC MEM1_STATUS
NOP  BUS REC MEM2_STATUS
```

3.0 PIPELINE CONSTRAINTS

As stated earlier, the 3230 CPU uses a four-phase pipeline structure. The word "pipeline" refers to the fact that instruction execution is implemented in phases and the fact that subsequent instructions are concurrently being executed in each of the phases. In the case of the 3230 CPU, four instructions are being executed concurrently, one in each of the four phases. A phase is one clock cycle long. During each clock cycle one instruction completes and exits the pipeline while an additional instruction begins or enters the pipeline. The three intermediate instructions advance within the pipeline. The four phases of the pipeline are:

Phase	Function
I	Instruction Fetch and Location Counter Selection
II	Instruction Interpretation and Operands Fetch
III	Arithmetic/Logical/Shift Execution
IV	Conditional Transfer/Skip Execution/Bus Initiate

The 3230 CPU composite instruction has two major sub-instructions, namely, the LHS portion (T:= A op B) and the RHS portion (C,Address). The following description of the various phases traces the effects of the two subinstructions as they pass through the pipeline. As a general rule, the LHS portion exhibits less variability than the RHS portion.

3.1 PIPELINE PHASE I

During Phase I, the location counter is used to address the Instruction memory and the instruction referenced is fetched. A clock transition terminates each pipeline phase. For Phase I the clock transition captures and holds the instruction (and its location counter value) for subsequent use in Phase II and simultaneously selects the new value for the location counter. Possible sources of new values for the location counter are:

(1) the current location counter value incremented by one,
(2) a transfer address from Phase II,
(3) a return address computed in Phase II,

(4) a conditional transfer address set by Phase IV.Note that the operation of Phase I cannot in any way affect its successor instruction. Only conditions in earlier phases (Phase II or IV) can affect the selection of the instruction subsequent to that in Phase I. The significance of this last statement can be more clearly appreciated after reading 3.2 and 3.4 below.

3.2 PIPELINE PHASE II

With regard to the assignment portion of the instruction, Phase II decodes the OP, A, and B fields and based upon the values of A and B reads out the values of the referenced registers or literals. At the terminating clock transition of Phase II, a decoded OP code is captured and passed to Phase III along with two 32-bit-wide values as selected by A and B. The T field is not decoded by Phase II but is merely passed along unaltered to Phase III.

The RHS instruction (C field) is decoded in Phase II, and the Unconditional Transfer, the Link, the Link Conditional, the Return, and the Load K Register are acted upon. All other instructions are simply passed along to Phase III.

For the Unconditional Transfer instruction, the address field is selected as the new value of the location counter (unless a selection is concurrently being made in Phase IV which takes precedence). This affects not the instruction in Phase I but rather the choice of the instruction which will follow it. Thus the Unconditional Transfer does not select its immediate successor instruction but rather the instruction following that.

The Link and Link Conditional instructions are like the Unconditional Transfer instruction but also "push" the accompanying location counter value onto the Link Stack Register (LSR) for subsequent use by the Return instruction. Like the Unconditional Transfer instruction, the Link instructions select the instruction following their immediate successor.

The Return instruction "pops" a value from the Link Stack Register (LSR) and adds the value of the address field to it. This sum then becomes the new value of the location counter (unless a selection is

concurrently being made in Phase IV which takes precedence). The Unconditional Transfer instruction, the Return instruction selects the instruction following its immediate successor.

The Load K Register instruction loads the 12-bit address field into bits 11-0 of the K register and sets bits 15-12 to zeros. This action takes precedence over a concurrent store into the K register from Phase IV (see 3.4).

3.3 PIPELINE PHASE III

Phase III acts exclusively on the LHS portion of the instruction and the transfer/skip portion (C field) is simply passed along to Phase IV. The 32-bit values selected in Phase II are combined or modified as specified by the OP field and the result is captured by the terminating clock transition (unless inhibited by Phase IV) in the X register.

The result of Phase III is not stored back into a general or special register immediately in Phase III but rather subsequently during Phase IV. Thus the assignment instruction R4:=R3+R2 followed by R5:=R4+R1 does not form the sum of R1+R2+R3 in R5 but rather forms the the sum of R1+R4 in R5, R4 contributing whatever original value it held when the two instruction sequence started. However, if a one instruction delay is imposed between the two instructions, the result is as expected. To achieve the desired result in two instructions, they should be written as R4:=R3+R2 followed by R5:=X+R1 in which case the sum R1+R2+R3 will appear in R5.

In summary, when the X register is specified, it contains the result of the preceeding instruction. However, a general register (R0 through R7) is always one cycle behind. If properly applied, this behavior can be used to advantage in programming the machine.

3.4 PIPELINE PHASE IV

During Phase IV, the result of the assignment statement formed in Phase III and held in the X register is stored in a general-purpose or special-purpose register if specified by the T field. The storage takes place in effect during the cycle rather than at its end.

Therefore, the result is available to the Phase II decode and operand fetch as if the storage had taken place at the transition of the clock of Phase III.

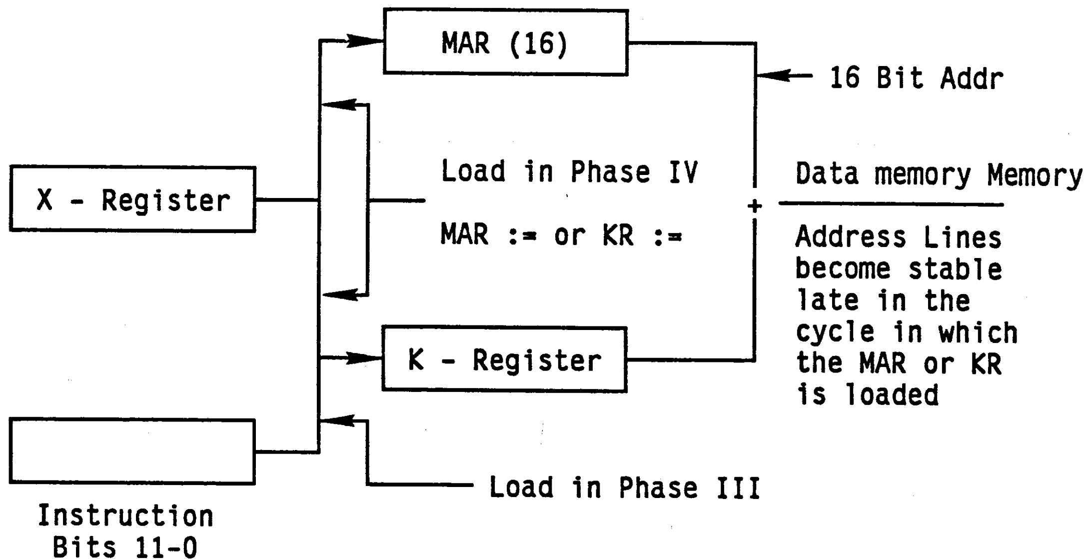
The store operation into the Interrupt Mask Register (IMR) requires special consideration. While storage into the IMR takes place during Phase IV, an additional clock cycle delay is required to develop the interrupt priority value (if any). Therefore, a minimum of two instructions must be imposed between storing into the IMR and the execution of a subsequent Link Conditional Instruction.

The result held in the X register at the end of Phase III is used during Phase IV for conditional transfer or skip testing. If the result of the test is true, then the address portion of the instruction is selected as the new value of the location counter and takes precedence over apotential unconditional transfer being concurrently executed in Phase II. It is also necessary to inhibit the instruction operation in Phase I. The instructions in Phase II and III must also be aborted. This inhibiting of the Phase I, II, and III instructions is referred to as "flushing the pipeline" and causes a loss of three instruction cycles. This loss only occurs for conditional instructions that actually transfer control. Thus good programming practice dictates that the conditional transfer be used to transfer on the least frequent or least probable logical path in order to reduce lost instruction cycles.

The Skip instructions operate like the conditional transfer instructions. The effect is to skip from one to four instructions immediately following the Skip instruction. Since the inhibited instructions actually pass through the pipeline (without effect), there is a one cycle loss for each instruction skipped. It takes as long to skip the instructions as to execute them, or to transfer past them.

3.5 PROGRAMMING/TIMING RULES FOR READING OR WRITING Data memory MEMORY

Proper reading and writing of data memory memory depends on assuring that addresses supplied via MAR and K have stabilized before supplying input for data memory writing via MIR. The following examples illustrate valid and invalid instruction sequences.



3.5.1 WRITE Data memory SEQUENCE

Valid

MAR := (value)
MIR := (value)   LDK (value)

In this example the MAR and KR are loaded in the same clock cycle. The memory address lines are stable for the following Write (MIR) to the data memory memory.

The MAR and /or KR may be loaded any number of instruction earlier.

Invalid

```
MAR := (value)
...
...
MIR := (value)
NOP    LDK (value)
```

The memory address lines will not be stable during this write operation and a mis-write may occur since the LDK and MIR are occuring during the same clock cycle.

3.5.2 READ Data memory SEQUENCE

Valid

```
MAR := (value)
NOP := LDK (value)
NOP
X := MOR + ( )
```

In this example the MAR and KR are loaded in the same clock cycle and become stable within that cycle. The subsequent MOR reads the memory output which takes time (15-25ns) to appear following the address lines becoming stable.

The MAR and/or KR may be loaded any number of instructions earlier.

Invalid

```
MAR := (value)
NOP := LDK (value)
X := MOR + ( )
```

The memory address lines will not be stable during operation and a mis-read may occur.

4.0 SPECIAL PROGRAMMING TECHNIQUES

This chapter describes special ways to exploit the 3230 CPU instruction set in performing operations common to procedural logic. In most cases the special techniques exploit the subtle features of the pipeline to achieve highest efficiency. Not all instruction effects are discussed in each example, only those effects pertinent to the example. Section 3 describes fully the phrasing of instructions.

4.1 LOOPS

To avoid cycle loss, unconditional transfers from the bottom to the top of a loop should be used for loop cycling, where the transfer is the next-to-last instruction in the loop, as illustrated below:

```
LOOP    <first instruction>
...
...

<LHS> TRA LOOP

<last instruction>
```

Since the unconditional transfer occurs in phase II of the pipeline, the <last instruction> is already in phase I. Thus it proceeds to completion while the transfer is taking place and no machine cycles are lost.

4.1.1 BOTTOM TEST LOOPS

A bottom-test loop may be implemented using a decrement and conditional transfer as the last loop instruction:

```
LOOP    <first instruction>
     ...
     ...
<LHS>  TRA LOOP
KR:=KR-1 TRA @+1 ON Z
```

Whenever the transfer condition is satisfied, the transfer to @+1 executes after unconditional transfer to LOOP and supersedes at exiting the loop. Three cycles are lost upon the exit, but no cycles are lost during looping.

4.1.2 TOP TEST LOOPS

A top-test loop may be implemented by using the decrement and conditional transfer as the first statement of the loop as follows:

```
LOOP    R0:=R0-1 TRA LOOPEND ON Z
     ...
     ...
<LHS> TRA LOOP
<last instruction>

LOOPEND <instruction>
```

Again, the conditional transfer loses cycles only if the transfer is made. Three cycles are lost upon loop exit, but no cycles are lost during looping.

4.1.3 ARRAYS: Data memory LOOPS

The use of the K register as a loop counter is particularly efficient for accessing arrays in the Data memory memory. The following example moves an array of length {R0} from location {R1} to location {R2}.

```
MOVE  KR := KR-1     TRA ENDMOVE ON HB    .decrement KR and test
                                           for finished
      MAR := R1+0                         .base address for
                                           fetch
      MAR := R2=0    TRA MOVE             .base address for
                                           store and loop
      MIR := MOR=0                        .copy data
```

Since the memory address referenced by the MOR and MIR is defined by the sum of the MAR and the KR, the KR is used as the common index of both fetch and store, and the MAR is used as a base register to alternate between the base address for fetching and the base address for storing.

4.2 OUT OF SEQUENCE EXECUTION

Often it is useful to select and execute a single instruction from an out-of-line table. The following table might represent alternative instructions to be selected based upon a computed value:

```
TABLE  X:=RO SLL 5
       X:=RO SLR 3
       X:=RO SLL 7
       NOP TRA ERROR4
       X:=RO SRL 2
```

The code for selecting and executing one of these instructions could be the following:

```
LSR:=R6+R1           .compute TABLE index
<instruction>        .wait for LSR value
<instruction>        .wait for LSR value
<LHS> RETURN TABLE   .execute selected instruction
<LHS> TRA @+1        .continue in sequence
```

By pushing the TABLE index into the LSR, the RETURN statement, with the TABLE address as argument, may be used to transfer into the TABLE. But since the TRA @+1 statement, following the RETURN, is already in the pipeline when the RETURN is executed, it will execute immediately after the selected table instruction enters the pipeline. Likewise, the selected TABLE instruction will execute in sequence behind the unconditional transfer actually overlapping the transfer in the manner that the transfer overlapped the return.

Note that at least two instructions must appear between the LSR stacking instruction and the RETURN. Otherwise the index value will not yet be available in the LSR when the RETURN pops the LSR for addition to the TABLE address.

4.3 TRANSFER VECTOR BRANCHING

An efficient transfer vector may have the following form.
Both the transfer and the instruction following are executed when this form of vector is used.

```
TRAVEC <LHS> TRA BRANCH1      .transfer 1
       <instruction>          .overlap transfer 1
       <LHS> TRA BRANCH2      .transfer 2
       <instruction>          .overlap transfer 2
       <LHS> TRA BRANCH3      .transfer 3
       <instruction>          .overlap transfer 3
```

The computed transfer corresponding to this vector is the following:

```
LSR:= R0 SLL 1          .multiply R0 by 2
<instruction>           .wait for LSR value
<instruction>           .wait for LSR value
<LHS> RETURN TRAVEC     .execute transfer
```

A more efficient transfer vector in space but less efficient in time is the following:

```
TRAVEC X:=0+0 TRA BRANCH1 ON Z
       X:=0+0 TRA BRANCH2 ON Z
       X:=0+0 TRA BRANCH3 ON Z
              <instruction>
```

Its corresponding computer transfer is the following:

```
LSR:=R0
<instruction>           .wait for LSR value
<instruction>           .wait for LSR value
<LHS> RETURN TRAVEC     .execute transfer
```

Note that the instruction following the TRA BRANCH3 is executed when that branch is taken.

4.4 REGISTER SHARING

The following sequence of code illustrates register sharing between instructions in the pipeline:

```
R5 := KR + 1

R5 := 0 + X'FFF'

X := R5 AND 3

R4 := R2 XOR R5 TRA ERR ON NZ
```

The arrows indicate the availability of the particular values of R5, and that two successive instructions access different values from R5. This same availability of data from the micro data memory applies to use of MIR and MOR.

```
MAR := 0 X'100'

MIR := 0 + 100

MIR := 0 + 101     LDK 0

MIR := MOR + 2     LDK 1

NOP                LDK 2
```

On occasion it is desirable to "borrow" a register for computation. This can be done by taking advantage of the storage time cycle as illustrated in the following example:

```
R5 := R6  SLL 2   <RHS>   .borrow R5
R5 := R5 + 0              .restore R5 to previous

X := MOR + R5
```

In this example R5 is altered temporarily as a value to apply in the third instruction, and restored to its previous value by the second instruction. In other situations X might have been used, but both X and MOR are A field-only addresses.

4.5 ACCESSING PRESTORED DATA FROM Instruction memory3

A table of constants may be prestored in a microprogram using the DC pseudo instruction as follows:

```
TABLE  DC <value1>
       DC <value2>
       DC <value3>
```

Values may be fetched from this table as follows:

```
LSR:=R5 OR 7          .set index into TABLE
<instruction>         .wait for LSR value
<instruction>         .wait for LSR value
<LHS> RETURN TABLE    .jump to "instruction"
X:=DATA+R3 TRA @+1    .add R3 to value
R3:=X+R2              .add R2 to value from X
```

Since the normal use of the DATA register is to access the subsequent instruction as a constant, e.g.,

```
X:=DATA+R3
DC <value>
```

the effect of the RETURN TABLE places the selected DC instruction from TABLE immediately following the

```
C:=DATA+R3 TRA @+1
```

and the TRA @+1 reinstates the normal execution sequence.

4.6 STACKING SUBROUTINE PARAMETERS

The LSR may be used to store one or two sixteen-bit parameters for use by a called subroutine. The sequence:

```
LSR:=0+VALUE1 LINK SUBROUTINE
LSR:=0+VALUE2
```

puts VALUE1 and VALUE2 on the stack after the return address, so the subroutine can pop them.

5.0 MEMORY INTERFACE

This section describes the interface from the 3230 CPU to the large user memory on the MAX 2. Data and control information are moved using the BUS EMIT and BUS RECEIVE instructions with appropriately coded information in the address field. Four independent channels for memory transfer are provided.

5.1 MEMORY CONTROLLER INTERFACE

The MAX 2 memory controller provides comprehensive controls over the flow of information to and from the 3230 CPU CPU. Overall structure is shown in Figure 7, below.

There are four independent channels to memory each with independent address registers and modes controlling access to memory. These channels continue to supply information information according to the modes and address specified until reset with new parameters. The channels are well adapted to applications in language interpretation and vector processing. For example, in vector processing one channel can carry the instruction stream while two carry vector source data and the fourth carries the result of the vector operation.

Memory addressing is to a single continuous linear address space. Memory size ranges from 1 to 64 megabytes. Two and four way interleaving is provided. The board uses 256K-bit chips in the 1-4 megabyte range, 1 mbit chips in the 4-16 megabyte range, and 4mbit chips in the 16-64 megabyte range.

Addressing modes are provided for bytes, halfwords, or words. Byte ordering is controllable in either the 0123 or 3210 order and halfword ordering may be 01 or 10. Addresses are incremented or decremented following each fetch according to a control mode. There are two elements of address incrementing: stride value and tiptoe value. Stride value is the major increment between element fetches. Tiptoe counts subelements, for example the two words for a double precision real or the four elements of a double complex element. Values of stride may be set to provide for row element access to a column stored array.

As shown in the figure below, the memory controller includes four identical channels each of which contains four registers which interface the user memory to the 3230 CPU. Information passes from the 3230 CPU X register through the controller to user memory, and data is fetched from user memory into the controller and from there read into EIR. (The term "fetched" is used for data movement from user memory to controller and the term "read" is used for data movement from controller to 3230 CPU.)

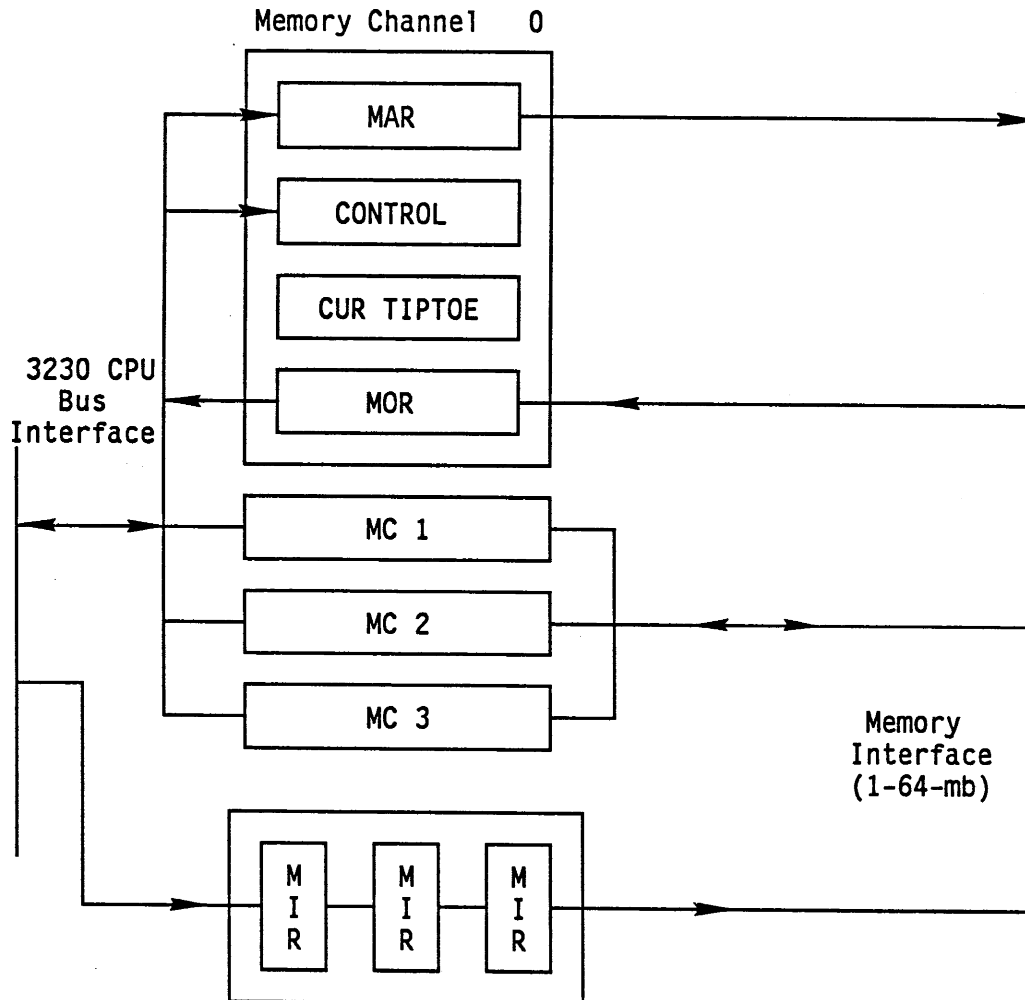
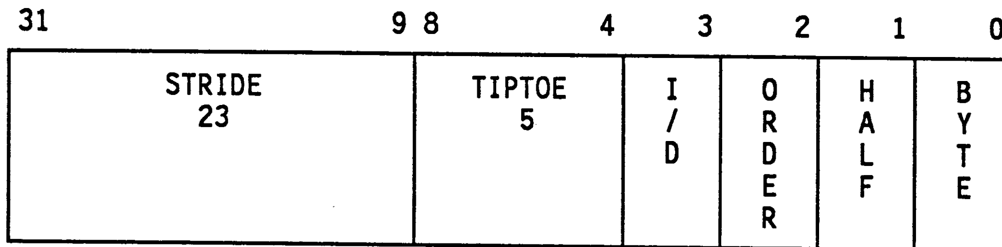


FIGURE 7 - Memory Controller Architecture

In each channel the four registers are:

1. MAR - memory address register
2. Control register with contents detailed in figure 8
3. Current Tiptoe register which counts sub-element fetches
4. MOR - memory output register which holds data fetched from memory until it is read by the 3230 CPU

A three deep memory input stack holds data waiting to be written to memory. These data may have come from any channel and are queued in the order written. If the queue fills, the 3230 CPU waits on the BUS instruction.



BYTE - Byte Indexing
HALF - Byte/Half Indexing
ORDER - Byte/Half Ordering: 0123 or 3210
I/D - Increment or Decrement
TIPTOE - Tiptoe Element Size
STRIDE - Twos Complement Index to Next Element

FIGURE 8 - Channel Control Word

Controller instructions are encoded into the address of the BUS EMIT/RECEIVE instruction described in section 2.6.8. The values for these operations are given in the standard definitions file, and listed in Appendix A.

5.1.1 BUS EMIT OPERATIONS

MEM_LDA - Load Address
This operation loads the controller address register from the X register.

MEM_LDF - Load Address and Fetch

This operation loads the address register and causes that addressed word to be fetched from extended memory into the controllers register.

MEM_LDFI - Load Address, Fetch and Increment

Same as MEM_LADRF except that in addition the address register is indexed according to the control register specification following the fetch.

MEM_WRT - Write Data

This operation transfers data from the X register into the controller's Write register and from there into the memory location indicated by the address register.

MEM-WRTI - Write Data and Increment

Same as MEM_WRT except that, in addition, the address register is indexed by one following the write.

MEM_SET - Set

Sets the control register of the channel with the contents of the X register.

5.1.2 BUS RECEIVE OPERATIONS

MEM_RD - Read

Data from the channel's MOR is moved to the EIR.

MEM_RDF - Read and Fetch

Data from the channel's MOR is moved to the EIR. In addition a fetch of information from memory is initiated.

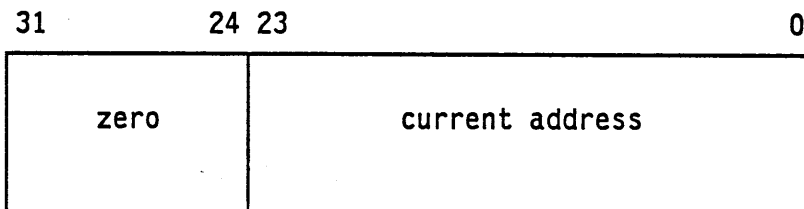
MEM_RDFI - Read Data, Fetch, and Index

Data from the channels MOR is moved to the EIR. In addition, a fetch of data from user memory is initiated, and then the address register is indexed according to the control register specifications.

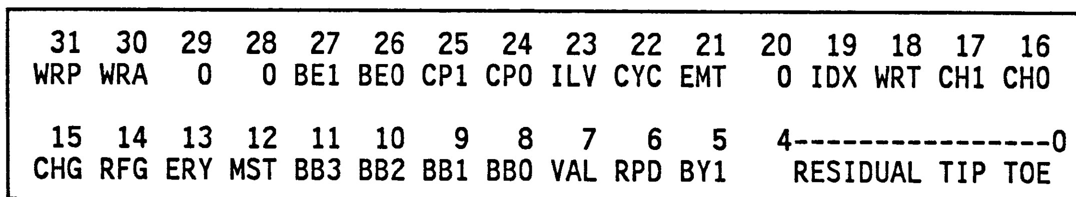
MEM_STAT1 - Read Memory Status1
Data from the controller's Status1 register is moved to the EIR. (See Figure 9)

MEM_STAT2 - Read Memory Status2
Data from the controller's Status2 register is moved to the EIR. (See Figure 9.)

STATUS1



STATUS2



CONTROLLER STATUS

WRP = THREE MEMORY WRITES PENDING
WRA = ANY WRITE PENDING
BE1,BE0 = 1 BANK(0), 2 BANKS(1), 4 BANKS(3) INSTALLED
CP1,CP0 = 256K RAMS(0), 1 MEGABIT RAMS(1), 4 MEGABIT RAMS(2)
ILV = INTERLEAVE ENABLED
CYC = MEMORY CYCLE IN PROGRESS
EMT = FIFO IS EMPTY
IDX = INDEX THIS CYCLE
WRT = WRITE CYCLE

CH1,CH0 = CHANNEL FOR CURRENT CYCLE
CHG = CHANNEL CYCLE,
RFG = REFRESH CYCLE
ERY = EARLY REFRESH
MST = MUST REFRESH
BB(3-0) = BANK 3,2,1,0 BUSY

CHANNEL SPECIFIC STATUS

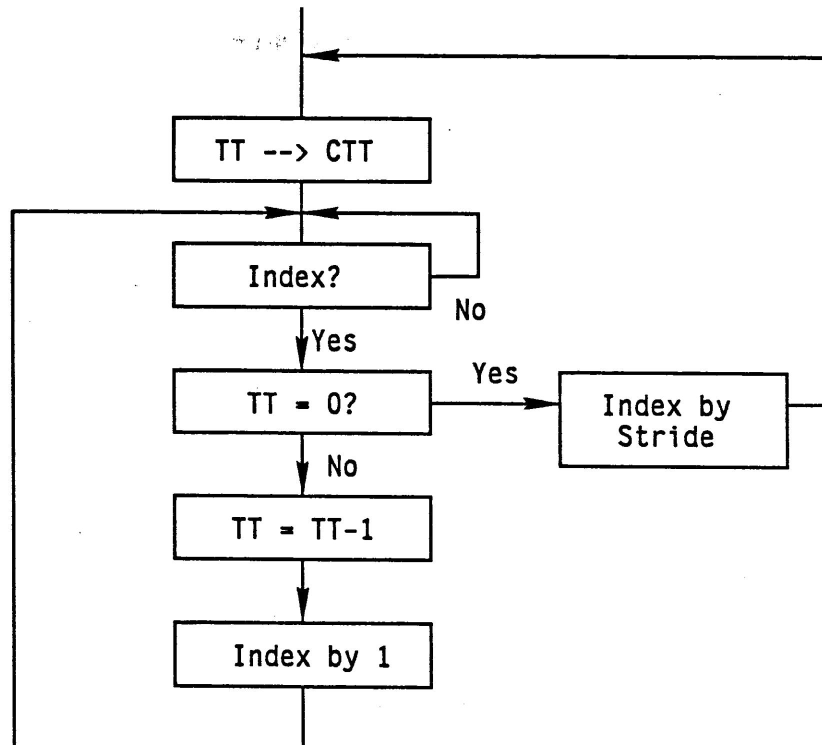
VAL = Data memory VALID
RPD = READ PENDING
BY1 = INDEX BY ONE

FIGURE 9 - STATUS WORDS

5.1.3 INDEXING

Indexing is carried out using the stride value, the tiptoe value and increment/decrement flag. The index addresses a byte address, a half-word address, or a word address depending on the settings of the byte and half-word flags of the control register. If the command given the controller requires indexing then it is carried out counting the tiptoe value to zero first and then indexing by the stride value.

Indexing means either add or subtract depending on the increment/decrement flag. Note also that the stride value is a twos-complement number which may have a plus or minus value. The flow chart below shows how stride and tiptoe control indexing.



5.1.4 TYPICAL PROGRAMMING

To read user memory into the 3230 CPU:

```
X := 0 + MEM_BYTE_MD+MEM_HL+MEM_UP+1*MEM_STRD+1*MEM_TT.
           .Setup control word for byte mode, high-to-low byte
            ordering, stride of one and tiptoe of one.
X := X + 0 BUS EMIT MEM_SET + MEM_CHO   .set control for
                                         channel zero
     ...
LOOP <LHS> BUS REC MEM_RDFI   .pick up next word, fetch and
                               index
           NOP
           NOP                .delay for pipeline
           R7 := EIR + 0      .memory word to R7
     ...
     <LHS> TRA Loop  .loop back for more
```

6.0 PC I/O AND SUPPORT INTERFACES

The MetaMicro Support Chip provides a 16-bit-wide data path between the 3230 CPU and the bus of an IBM PC-AT or one of its clones. In addition, the support chip provides for loading of the writeable control store, processing data from the scan-out logic, and controlling various logical functions such as single-step mode and RUN mode, breakpoint detection, and reset functions.

The support chip has two interfaces, one directed to the MetaMicro and one directed to the PC-AT bus. In addition, discrete logic on a MAX 2 board is present to provide features not directly incorporated into the support chip.

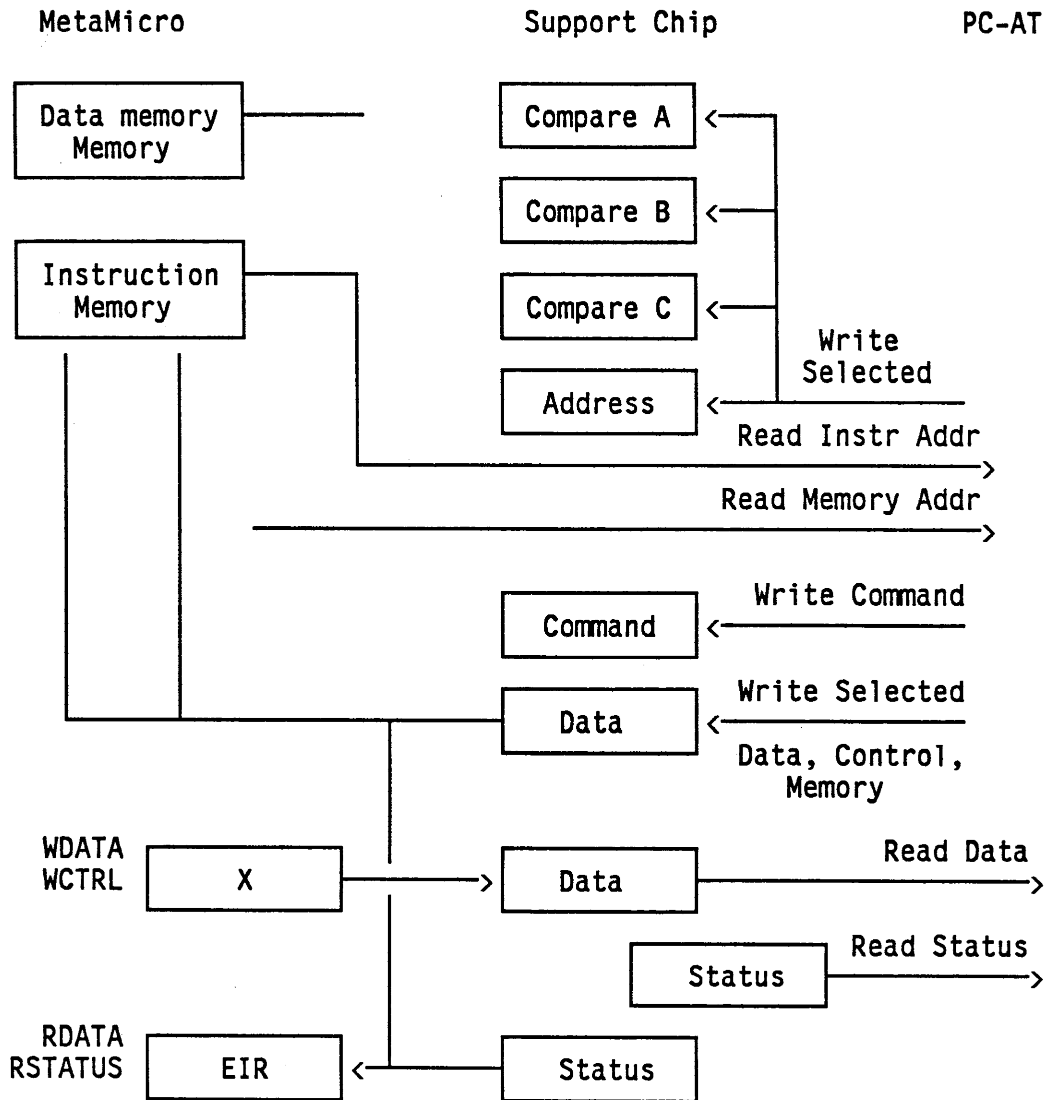


FIGURE 7 - Support Chip Overview

From the perspective of the 3230 CPU, the support chip resides on the external bus (Figure 1) and responds to BUS EMIT/REC commands under control of the microprogram. There are two read and two write commands. Data to be read or written are transmitted on bits 15-0 and associated parity bits. Data transferred via bus bits 31-16 are ignored and data received via bus bits 31-16 will have a zero value.

Information is transferred between MetaMicro and PC via two data registers in the support chip, one register for each direction of transfer. Transfer is syncronized through status bits which are set when information is deposited in the data register and reset when the information is picked up. Four bits are involved: a pair for each direction of transfer between MetaMicro and PC, a bit of each pair indicates whether the information is data or control. They are set when the register is loaded and reset when information is picked up on the other side. Operation of these status bits depends on the direction of transfer.

6.1 METAMICRO TO PC

If a write is given before previously written data is picked up from the data register, or if a read is given before the data register has been loaded from the PC side the BUS instruction waits. After 16 cycles I/O timeout (ITMO) is initiated setting the timeout interrupt status bit.

A normal programming technique for transfers to and from the PC is to examine status before issuing the read or write instruction in a "spin" loop waiting for the proper setting of the status bits.

6.1.1 ACCESS

The Support Interface is accessed using standard externalbus addressing rules as described in the table below:

INPUTS FROM THE EXTERNAL BUS

Mnemonic	Address*	Function
PC WDATA	PC-ADDR+1	Write Data
PC WCTRL	PC-ADDR+0	Write Control

OUTPUTS TO THE EXTERNAL BUS

Mnemonic	Address*	Function
PC RDATA	PC-ADDR+1	Read Data Register
PC RSTATUS	PC-ADDR+0	Read Status

* "PC-ADDR" refers to subsystem address on the MAX 2 bus, bits 5-1 of the six-bit bus address. The value of "PC-ADDR" on the MAX 2 is binary X'20'.

6.1.2 STATUS TO METAMICRO

The format and meaning of the status bits from the perspective of MetaMicro is given below:

Position	*Meaning*
Bit 0	*Data Available from PC/AT to MetaMicro*
Bit 1	*Control Available from PC/AT to MetaMicro*
Bit 2-7	*Not Used (Received as Zero)*
Bit 8	*Data Available from MetaMicro to PC/AT*
Bit 9	*Control Available from MetaMicro to PC/AT*
Bit 10	*Valid Parity on Data/Control from 3230 CPU to support chip (both byte 0 and byte 1)*
Bit 11	*Not Used (Received as Zero)*
Bit 12	*Valid Parity on Data/Control from 3230 CPU to support chip (byte 0)*
Bit 13	*Valid Parity on Data/Control from 3230 CPU to support chip (byte 1)*
Bits 14-15	*Not Used (Received as Zero)*

6.1.3 READ FUNCTIONS

The "Read Data Register" function causes the interface to send to the 3230 CPU the contents of a 16-bit Data Register previously loaded by actions of the associated PC-AT. The data in the "Data Register" were loaded by the PC-AT as either "data" or "control" information as recorded in the status register. The PC-AT MetaMicro data/control bit is reset as a result of the "Read Data Register" function. If neither the "data" nor the "control" bit is set, the function is deferred until timeout.

The "Read Status" function caused the support chip to send to the 3230 CPU the contents of the status register formatted as described below. The "Read Status" function is never deferred.

6.1.4 WRITE FUNCTIONS

The "Write Data" function transmits 16 bits from the X register to the interface and sets the "data" bit in the status register if neither the "data" nor "control" bit were previously set. If either bit was set, the operation is deferred until timeout.

The "Write Control" function transmits 16 bits from the EIR to the interface and sets the "control" bit in the status register if neither the "data" or "control" bit were previously set. If either bit were set, the operation is deferred.

The presence of the "data" or the "control" bit in the status register initiates an interrupt to the PC-AT. PC-AT interrupt 10, 11 or 12 or none is used depending on jumper wiring on the Metaframe board.

6.2 INTERFACE TO PC-AT

From the perspective of the PC-AT, the support chip contains several 16-bit registers which can be written to and several 16-bit registers which can be read. It is addressed from the PC-AT as a "prototype" input/output device with the following address structure:

Address Bits

9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*(Address bit position)*
1	*1*	*0*	*0*	*0*	S_1	S_0	*0*	*F*	*0*	*(Bit Value)*

Where:

S_1 and S_0 are established by jumper wiring on the MAX 2 board and permits up to 4 MAX 2 boards to be placed in a PC-AT.

F is a function bit (0 or 1) and is used to address the various read/write operations of the support chip.

6.2.1 PC TO METAMICRO

For this side of the transfer there is no automatic waiting on the status bits as is the case for the 3230 CPU to Support interface. When the PC writes, information enters the data register regardless of the setting of the status bits and a read information is always transferred. Thus the program on the PC side must syncronize by examining the status bits before issuing a read or write.

6.2.2 ACCESS

Accessing the PC-AT-SUP Interface is accomplished using standard PC-AT Input/Output commands, the basic I/O address described above, and the F bit of the I/O address. The available functions are described in table format below. The letters "BA" refer to the basic PC-AT I/O address above.

INPUTS FROM PC-AT BUS	
Address	Function
BA+(F=0)	Write Command Register
BA+(F=1)	Write Selected Register

OUTPUTS TO PC-AT BUS	
Address	Function
BA+(F=0)	Read Status
BA+(F=1)	Read Selected Register

6.2.3 READ STATUS FUNCTION

The "Read Status" BA+(F=0) transmits the contents of the status register from the support chip to the PC-AT formatted as described below.

6.2.4 STATUS TO PC-AT

The format of the status bits from the perspective of the PC-AT is:

Position	*Meaning*
Bit 0	*Data Available from MetaMicro to PC-AT*
Bit 1	*Control Available from MetaMicro to PC-AT*
Bit 2	*Valid Parity on Data/Control from 3230 CPU to support chip (both byte 0 and byte 1)*
Bit 3	*Not Used (Received as Zero)*
Bit 4	*Valid Parity on Data/Control from 3230 CPU to support chip (byte 0)*
Bit 5	*Valid Parity on Data/Control from 3230 CPU to support chip (byte 1)*
Bit 6	*Data Available from PC-AT to MetaMicro*
Bit 7	*Control Available from PC-AT to MetaMicro*
Bit 8-10	*Instruction Address Compare with Compare Registers C,B,A*
Bit 11	*Not Used (Received as Zero)*

Bits 14-12	*Memory Address Compare with Compare Registers C,B,A*
Bit 15	*Not Used (Recieved as Zero)*

6.2.5 WRITE COMMAND REGISTER

The "Write Command Register" BA + (F=0) transmits a 16 bit word from the PC-AT to a "Command Register" (CR) in the support chip. The contents of the CR establish various conditions such as clear for the support chip and for the MAX 2 or ProtoFrame PC board. In particular, the CR contains a 3-bit function field which serves as a register select address for the Write or Read Selected Register functions BA+(F+1). The format and meaning of the Command Register are described below:

Command Register Format

Position	*Meaning*
Bit 0	*Clear (=0), don't clear (=1)*
Bit 1	*Stop (=0), Run (=1)*
Bit 2	*Inst. Addr. Disable (=0), Enable(=1)*
Bit 3	*Scan-out Disable (=0), Enable (=1)*
Bit 4	*No Clock (=0), Clock Cycle (=1)*
Bits 5-7	*Function Select Field (FSF)*
Bits 8-10	*Instruction Compare Disable (=0), Enable(=1)*
Bit 11	*Not Used*
Bits 14-12	*Memory Compare w/C,B,A Disable(=0), Enable(=1)*
Bit 15	*Compare Interrupt Disable (=0), Enable (=1)*

Function Select Field: The Function Select Field (FSF) operates in conjunction with "Write Selected Register", "Read Selected Register" and bit 4 of the Command Register, the No Clock/Clock Cycle bit. In effect, the FSF specifies which register to write, which register to read or which clock pulse to generate. The FSF value and register/clock assignments are described in the table below.

FSF	Write	Read	Clock Pulse
0	Data	Data Register	Scan Clock
1	Control	Memory Address	Memory Write Clock 1
2	Memory	Inst. Address	Memory Write Clock 2
3	Inst. Addr.	Not Used	Step Clock
4	Compare A	"	Not Used
5	Compare B	"	"
6	Compare C	"	"
7	Not Assigned	"	"

FUNCTION SELECT FIELD ASSIGNMENTS

6.2.6 WRITE SELECTED REGISTER

"Write Selected Register" with FSF=0 transmits a 16-bit word from the PC-AT to the data buffer in the support chip and sets the "Data available from PC-AT to 3230 CPU" bit in the status register.

"Write Selected Register" with FSF=1 transmits a 16-bit word from the PC-AT to the data buffer in the support chip and sets the "Control Available from PC-AT to 3230 CPU" bit in the status register.

"Write Selected Register" with FSF=2 transmits a 16-bit word from the PC-AT to the data buffer in the support chip. The status register is not affected.

"Write Selected Register" with FSF=2-6 each transmit a 16-bit word from the PC-AT to the instruction address and the three compare registers A,B, and C, as shown in the table above. The status register is not affected.

6.2.7 READ SELECTED REGISTER

"Read selected Register" with FSF=0 transmits a 16-bit word from the Data Register to the PC-AT. Similarly FSF=1 and 2 respectively transmit the 16-bit contents of the Memory Address (3230 CPU MAR+K) and Instruction address registers to the PC-AT.

6.2.8 OTHER COMMAND REGISTER FUNCTIONS

Bit 0, clear (=0)/don't clear (=1). This function resets the data and control flags of the support chip and issues a clear signal to the 3230 CPU which resets interrupts, sets the instruction address to zero, and clears the pipeline.

Bit 2, Instruction Address Disable (=0)/Enable (=1). The 3230 CPU has an input signal IAEMB, which causes it to deliver a 16-bit instruction address to its instruction memory. In order to permit loading the instruction memory by the PC-AT, the address from the 3230 CPU must be turned off (disabled, bit 2=0) and an instruction address register in the support chip must be turned on. Thus, with Bit 2=0 the 3230 CPU address is turned off and the support chip address is turned on; with Bit 2=1 the 3230 CPU address is turned on and the support chip address is turned off.

Bit 3, Scan-out Disable (=0)/Enable (=1). The 3230 CPU has internal logic to gain serial (bit-by-bit) access to the value of certain 3230 CPU registers for diagnostic (hardware/software) purposes. They are the X register and the instruction in Phase II of the pipeline. Two input signals, SCLD (Scan-Load) and SCCLK (Scan Clock) and one output signal SCDATA (Scan Data) are present on the 3230 CPU processor. Matching signals of the same name are also present on the support chip.

The 3230 CPU Scan-Out functions as follows: Bit 3 of the Command Register is Enabled (=1). A Scan-clock (SCCLK) is generated (see below). This sequence of events causes registers of the 3230 CPU to be parallel loaded into an extended shift register (the Scan-Out Register) within the 3230 CPU. Bit 3 of the Command Register is then disabled (=0); the high-order bit of the Scan-Out Register is made available on the SCDATA output pin of the 3230 CPU for each subsequent Scan-Clock (SCCLK) that is generated the Scan-Out Register of the 3230 CPU shifts left one position. Thus, each succeeding bit of the Scan-Out Register becomes available on the SCDATA output pin with each Scan-Clock.

The mode of operation of the support chip and the 3230 CPU assumes that the SCLD, SCCLK, and SCDATA of the 3230 CPU are connected to the like named pins of the support chip. Within the support chip is a 16- bit

shift register designed to receive serial data from the 3230 CPU SCDATA output. With SCLD Low (disabled) each SCCLK pulse captures one bit of the 3230 CPU Scan-Out Register. After 16 SCCLK pulses the upper 16 bits of the 3230 CPU's Scan-Out Register have now been transferred to the support chip data register and can be read out, in parallel, by the PC-AT. The PC-AT thus gains access to the internal registers of the 3230 CPU.

Bit 4, No Clock (=0)/Clock Cycle (=1) A one bit written into bit 4 of the Command Register causes one clock pulse to be generated depending upon the value of the Function Select Field (Bits 7-5). Bit 4 of the Command Register is reset as a result of the clock pulse generation. For each clock cycle desired the PC-AT must rewrite command register bit 4.

7.0 FLOATING POINT COPROCESSOR

The floating point coprocessor (FPC) is an optional daughter board for the MAX 2 that provides single and double precision IEEE format floating point arithmetic, integer multiply and divide, conversion of integers, single and double floating point into one-another, and the functions SIN, COS, A**B, TAN, ATAN, and LOG.

The board includes ALU and MPY units, 4096 words of 48-bit memory for microcoding of the functions carried out by the FPC, and control and interface logic.

The 3230 CPU communicates with the FPC over the 32-bit external bus of the MAX 2 using BUS EMIT and BUS RECEIVE instructions. The FPC responds to bus address X'38'.

7.1 BUS EMIT AND RECEIVE FUNCTIONS

There are two levels of functions carried out by the FPC: 7 primary functions and an arbitrary number of secondary functions.

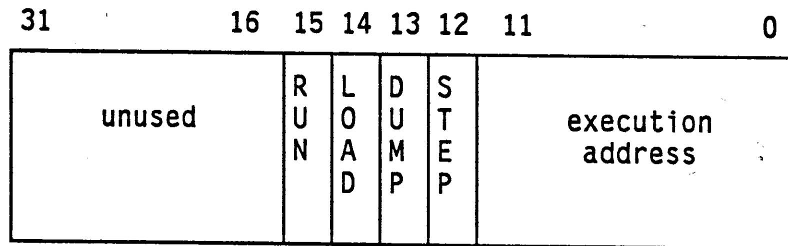
Primary functions are given in the address field of the BUS EMIT instruction which also transmits the first operand to the FPC. Primary functions are therefore faster than secondary functions.

The seven primary functions are:

1. Integer multiply
2. Single precision floating addition
3. Single precision floating subtraction
4. Single precision floating multiply
5. Double precision floating addition
6. Double precision floating subtraction
7. Double precision floating multiply

The eighth code causes the word transmitted over the bus to be used as a function initiation word rather than the first argument. This function initiation word provides the specific interpreter address to be executed on the FPC.

The function initiation word layout is:



The four control bits are set to mean:

RUN - execute the program at the address given
LOAD - write FPC interpreter memory
DUMP - read PPC interpreter memory
STEP - single step execute interpreter at address

The secondary function provided are:

1. Integer to single precision floating conversion
2. Single precision floating to integer conversion
3. Integer to double precision floating conversion
4. Double precision floating to integer conversion
5. Single to double precision floating conversion
6. Double to single precision floating conversion
7. Single precision floating division
8. Double precicion floating division

Current status of the coprocessor is determined using a BUS RECEIVE operation with address X'3A'. The status word returned has the following format:

31	30	29	28		22 16	15 10	9	8	7	6	5	4	3	2	1	0
RUN	WI	WO	HALT	---	address	---	OVR	UND	INX	INV	DEN	OVR	UND	INX	INV	RCO
							ALU				Multiply Unit					

Meaning of the various bits are:

```
 RUN = running
  WI = waiting for input
  WO = waiting for output
HALT = halted
 OVR = overflow
 UNO = underflow
 INX = inexact
 INV = invalid
 DEN = denormalized
 RCO = round carry out
```

7.2 SAMPLE PROGRAMS

This program multiplies the 32-bit real in R7 by that in R6.

```
X := 0 + R7     BUS EMIT   CP_FMPY    .Send 1st
                                       argument and
                                       command
X := 0 + R6     BUS EMIT   CP_OPER    .Send 2nd
                                       argument
<LHS>           BUS REC    CP_RSLT    .Read result
                                       from FPC
 NOP
 NOP                                  .Pipeline wait
R3 := EIR + 0                         .Result to R3
```

The code below divides the first double precision floating point argument by the second. A completion interrupt is used to wait for the result.

Double precision divide R = A/B

Input order: B(LSW),A(LSW),B(MSW),A(MSW)
Output order: R(LSW),R(MSW)
Completion interrupt: Yes
Error interrupt: Underflow, Overflow, Invalid, Inexact

Example:

```
     X:=0+CP_FPODIV                   .Setup routine address
     X:=X=0 BUS EMIT CP_INIT          .Initiate function
     X:=R1+0 BUS EMIT CP_OPER         .Send S(LSW)
     X:=R2+0 BUS EMIT CP_OPER         .Send A(LSW)
     X:=R3+0 BUS EMIT CP_OPER         .Send B(MSW)
     X:=R4+0 BUS EMIT CP_OPER         .Send A(MSW)
WAIT IR:=IR AND CP_CMPINT             .Check interrupt
     X:=X+0 TRA WAIT ON Z             .Wait for interrupt
     NOP
     NOP BUS REC SP_RSLT              .Get result (LSW)
     NOP
     NOP
     R2:=EIR+0                        .Store result (LSW)
     NOP BUS REC CP_RSLT              .Get result (MSW)
     NOP
     NOP
     R3:=EIR+0                        .Store result (MSW)
```

APPENDIX A: ASSEMBLER SYNTAX AND PSEUDO OPS

This appendix defines the syntax of the assembly language and describes the assembler pseudo operations. The assembler runs on the IBM PC-AT or compatible personal computers.

Syntax for the instructions is given in Secticn 2 of this manual. The language is designed to reflect the 3230 CPU architecture as closely as possible while maintaining readability and legibility. For example, expressions much like those found in higher level languages are allowed.

Finally, there is a command which may be used to change the keywords to accommodate individual programmer's style.

The assembler is called from the DOS prompt. There are three parameters on the call given the file names for 1) assembler source, 2) listing output, 3) run-time output. No linker is required; a directly loadable image is produced for the instruction memory. A typical assembler call is:

```
MMASM   MYSOURCE, MYLIST, MYRUN   [/ OPTION]
```

Parameters are separated by commas. If list and/or run unit files are not given, then no output of that kind is produced. Three options control EPSON or compatible print control: /C+ for compressed, /E+ for elite, /P+ for pica. The default is no printer control. For convenience several batch files are provided to produce often used assemblies.

A1 LANGUAGE ELEMENTS

A1.1 COMMENTS

A period causes the rest of the line to be treated as a comment. If the first nonblank character on the line is a period, or if the entire line is blank, then the whole line is assumed to be a comment.

```
<comment> ::= .  <char_string>
```

A1.2 IDENTIFIERS

Identifiers are used as labels and names. An identifier is a string of up to eleven characters, the first of which is a letter. Underscore characters may be included and all characters are significant, including underscores.

```
<identifier> ::= <letter>|
                 {[underscore](<letter> <digit>)}
```

Examples:

X	NAME	ENTRY_TO_A1
X1	LAB_X	LABEL___33

A1.3 NUMBERS

There are three classes of numbers: decimal integers,hexadecimal integers, and character strings. They can be represented as follows:

```
<number> ::= <dec_integer>|<hex_integer>|<char_string>
```

A1.3.1 DECIMAL INTEGERS

```
<dec_integer> ::= <dec_digit> {<dec_digit>}

<dec_digit> ::= 1 2 3 4 5 6 7 8 9 0
```

A1.3.2 HEXADECIMAL INTEGERS

```
<hex_integer> ::= X'<hex_digit> {<hex_digit>}'
<hex_digit> ::= 1 2 3 4 5 6 7 8 9 A B C D E F 0
```

A1.3.3 CHARACTERS

```
<char_string> ::= C'<char> {<char>}'
<char> ::= any ASCII character
```

Examples:

```
0            37801        16
X'0'         X'FF031A'    X'10'
C'ABCD'      C'12'        X'A D"'
```

<u>Notes:</u> A negative hex integer is indicated by -X'FF'; <u>not</u> by X'-FF'. The maximum allowable magnitude is $2^{31}-1$. Hence, X'7FFFFFFF' is the largest representable hex integer and 2147483647 is the largest representable decimal integer. The longest character string is four characters.

A1.3.4 REAL NUMBERS

```
<real_number> ::= R'<real number or integer><E integer>'
```

Examples:

```
R'3.5'
R'-3E12'
R'1.223E05'
```

A1.3.5 DOUBLE PRECISION NUMBERS

```
<double precision #>::= D'<real number or integer><D integer>'
```

Example:

```
D'3.5'
D'-1D12'
D'1.223E05'
```

A1.4 EXPRESSIONS

An expression is any well-formed algebraic expression involving identifiers, numbers, left and right parentheses, the operators +, -, *, /, &, |, ~, <, and > and the special symbol, @, which indicates the current value of the present location counter.

The usual rules of precedence found in FORTRAN and most other higher level languages apply. It is permissible to mix hexadecimal integers and decimal integers in the same expression. The exact definition follows.

```
<expr> ::= [-] <term> {(+|-)<term>}
<term> ::= <factor>{(*|/)<factor>}
<factor> ::= <primary> | (<expr>)
<primary> ::= <number> | <identifier>| @
```

Examples:

```
0                  A15*X'FF01'
X'0'               (-B*3/X) + (X'A3' - Z_AB)
15-X'A'            (((2*A) *3) - Z + X'2')
@                  @ + 256
R'3.5'             D'3.5'
R'3E12'            D'1D12'
R'1.223E05'        D'1.22E05'
```

Notes: Except for expressions used in EQU, RS, ORG, and COMWIDTH statements, the identifiers in an expression need not be defined before. The division used is integer division, and the result will be truncated without warning.

A1.4.1 LOGICAL OPERATORS

Four operators are used to perform logical functions on expressions. They are:

```
&    AND
|    OR
!    XOR
~    NOT
```

The & and | and ! operators have the same precedence as the * and / operators. The ~ operator has the same precedence as the + and - operators.

A1.4.2 SHIFT OPERATORS

Two shift operators shift an expression right or left a specified number of places.

Syntax:

expr > expr   for right shift
or
expr < expr   for left shift

Examples:

2<3 has the value 16
16>3 has the value 2

Shift operators have the same precedence as * and /.

A2 PSEUDO-OPERATIONS

The pseudo ops are divided into two classes: assembler listing commands (or those which do not affect object code) and assembler action commands (or those which affect object code).

<pseudo_op> ::= <action_pseudo_op>|<listing_pseudo_op>

A2.1 ASSEMBLER ACTION COMMANDS

The following pseudo operations are those that have a possible effect on the generated code.

<action_pseudo op> ::=

<label_stmt>| <change_stmt> | <mode_stmt>|
<org_stmt> | <define_const>| <reserve_store> |
<equate_stmt>| <insert_access> | <insert_stmt>|
<end_stmt> |

A2.1.1 LABEL (LONG LABEL DEFINITION)

Generally, any character string starting in column one is assumed to be a label. Occasionally, a statement containing one or more long expressions may require a label for which there is no space on the line. In this case, the statement may be preceded by a label statement.

<label_stmt> ::= LABEL <identifier>

Example:

```
LABEL     CONSTANT_10
          DC 10
```

Notes: Avoid using the LABEL statement to give a statement two or more labels, because an error will result. If it is necessary to give a statement a second label, an EQU statement should be used.

A2.1.2 CHANGE, UNCHANGE (KEYWORD ALTERATION)

On occasion, the user may find it convenient to change one of the built-in keywords by means of a change statement.

```
<change_stmt> ::= CHANGE <keyword> [TO] <char_string>|
                  UNCHANGE
```

Examples:

```
CHANGE HB TO HIGH_BIT=1
CHANGE := =
UNCHANGE
```

Notes: "TO" is optional unless it is the keyword "TO" which is being changed. UNCHANGE causes all keywords to revert to their original forms. Avoid using the change statement to introduce ambiguities into the language. The operators +, -, *, and / as used in expressions may not be changed. Keywords are limited to eight characters.

A2.1.3 INST, DATA (INSTRUCTION, DATA MODE INITIATION)

INST changes the mode of the location counter to "instruction", while DATA changes it to "data".

```
<mode_stmt> ::= INST | DATA
```

A2.1.4 ORG (SET ORIGIN OF ABSOLUTE CODE)

ORG assigns the value of the expression following ORG to the current location counter.

```
<org_stmt> ::= ORG <expr>
```

Examples:

```
DATA
ORG  256
...
INST
ORG @ + X'F'
```

A2.1.5 DC (DEFINE CONSTANT)

DC evaluates an expression and generates code containing its value.

```
<define_const> ::= DC <expr>
```

Examples:

```
TEN      DC      10
AB       DC      X'15' - AC
```

A2.1.6 RS (RESERVE STORAGE)

RS causes the current instruction counter to be incremented by the value of the expression following the RS.

```
<reserve_store> ::= RS <expr>
```

Example:

```
LAB      RS      15
```

Note: An error message is given for the first use of an RS statement while in INST mode.

A2.1.7 EQU (EQUATE SYMBOLS)

This statement must be used with a label. The expression following the EQU is evaluated and the label is entered in the symbol table with the resultant value.

```
<equate_stmt> ::= EQU <expr>
```

Examples:

```
LABEL      LAB_ONE
EQU  15 + X'78' * L5
HERE_PLUS_5  EQU @+5
```

Notes: The equate statement generates no code and has no effect on the location counters. Any identifier appearing in the expression of an EQU, RS, ORG, or COMWIDTH statement must be defined prior to use.

A2.2 ASSEMBLER LISTING COMMANDS

The following commands affect the program listing provided by the assembler but have no affect on the code generated.

```
<listing_pseudo_op> ::=

      HEADER <char_string> | FORMAT | NOFORMAT |
      COMWIDTH <expr> | BLOCKS | NOBLOCKS | EJECT |
      LISTC | NOLISTC | ILIST | NOILIST | WARN | NOWARN
```

A2.2.1 HEADER (PAGE HEADINGS)

HEADER prints the character string appearing on the remainder of the card at the top of each page of the listing until another header instruction is encountered.

A2.2.2 EJECT (PAGE EJECTION)

EJECT causes the printer to skip to the top of the next page.

A2.2.3 LIST, NOLIST (ASSEMBLY LISTING ON, OFF)

LIST causes the assembler to begin listing subsequent statements. NOLIST supresses listing. The default is LIST.

A2.2.4 LISTC, NOLISTC (LISTING COMMANDS ON, OFF

LISTC causes the assembler listing commands to be listed. NOLISTC supresses listing them. The default is NOLISTC.

A2.2.5 BLOCKS, NOBLOCKS (BLOCK STRUCTURE ANNOTATION CONTROL)

BLOCKS causes the assembler to indicate the structure of the logical blocks of the source program. Right arrows are provided in the listing for each labeled instruction and left arrows for each transfer, link, or return instruction. The default option is BLOCKS; NOBLOCKS turns off this feature.

A2.2.6 FORMAT, NOFORMAT (FORMATTED LISTING CONTROL)

The formatter will line up the labels and various fields of the source statements before printing them. Also, labels defined by the label pseudo_op are printed on the correct lines. FORMAT turns on the formatter, while NOFORMAT turns it off. The default is FORMAT.

Note: Be careful when using CHANGE and FORMAT at the same time. In the format mode, it is assumed that the following maximum field widths are used:

```
T_field      - 4 characters
A_field      - 4 characters
:=(assign)   - 2 characters
op_field     - 3 characters
```

If any of these fields use more than the allotted length, the excess will be ignored, resulting in possible syntax errors. (This will never happen if the change operation is not applied to any of the involved fields.)

A2.2.7 COMWIDTH (COMMENT WIDTH FOR JUSTIFICATION)

Comment width. The formatter attempts to right justify comments while lining up their left hand sides. COMWIDTH is used to specify the expected length of comments; the default value is 45.

A2.2.8 ILIST, NOLIST (INCLUDE FILE PRINTING ON, OFF)

ILIST causes printing of statements from include files. NOILIST suppresses printing. The default is NOILIST.

A2.2.9 WARN, NOWARN (WARNING MESSAGE PRINTING ON, OFF)

WARN causes warning message to be printed, NOWARN suppresses printing. The default is WARN.

A2.2.10 END (END OF ASSEMBLY)

END signals the end of the assembly source. END terminates the assembly. All input beyond the END command is ignored.

A2.2.11 CODELEN

CODELEN specifies the largest number of 32-bit words of code the program can create. The default is 4096 words. The maximum is 16384.

Syntax:

```
CODELEN   <expr>
```

Example:

```
CODELEN   4096      The default.
CODELEN   200       The maximum code length is
                    200 words.
```

A2.2.12 IFON, IFOFF

IFON causes all conditional code in the #IF #ELSE #END blocks to be printed in the listing. IFOFF causes only the code that is to be assembled to be printed. The default is IFON.

A2.3 DEFAULT OPTIONS

The status of the assembler upon initialization of execution is the same as if it had just read the following sequence of instructions.

```
LIST
NOLISTC
EJECT
HEADER
FORMAT
CODELEN 4096
COMWIDTH 50
DATA
ORG 0
INST
IFON
NOILIST
WARN
NOBLOCKS
```

A3 PRE-PROCESSOR COMMANDS

There are several PREPROCESSOR COMMANDS that change the stream of code to be assembled. All of these commands start with a '#' character in column 1.

A.3.1 #IF #ELSE #END

Syntax:

```
#IF <expr>
...
body of code
...
#ELSE
...
body of code
...
#END
```

The body of code located between the #IF statement and the #ELSE statement is assembled if the value of expression is non-zero. Otherwise the code between the #ELSE and the #END statement is assembled. The #ELSE statement is optional.

A.3.2 #INCLUDE

Syntax:

```
#INCLUDE filename
```

The code contained in the file specified is compiled in line with the rest of the code. The filename may be any DOS filename including the path, if needed.

A.3.3 #DEFINE

Syntax:

```
#DEFINE   label   <expr>
```

The #DEFINE command assigns the value of the expression to the label. The #DEFINE is similar to the EQU, only the #DEFINE command may redefine the value assigned to a label as often as needed.

A4 STANDARD ASSEMBLER MNEMONICS

Standard mnemonics and bit assignments used in IMS code are given in the table on the next page. These values are also provided as a file on the the assembler release diskette.

```
CODE        STATEMENT
            .............
            .............  STANDARD SKIP MNEMONICS AND BIT ASSIGNMENTS
            .............
00000001    NOV            EQU  X'01'      .NO-OVERFLOW
00000002    OV             EQU  X'02'      .OVERFLOW
00000004    NLB            EQU  X'04'      .NO-LOW-BIT
00000008    LB             EQU  X'08'      .LOW-BIT
00000010    Z              EQU  X'10'      .ALL-ZERO-RESULT
00000020    NZ             EQU  X'20'      .NON-ZERO-RESULT
00000040    NHB            EQU  X'40'      .NO-HIGH-BIT
00000080    HB             EQU  X'80'      .HIGH-BIT
            .............
            .............  HARDWARE ADDRESSES
            .............
00000020    META_PC        EQU  X'20'      .META_MICRO/PC INTERFACE
            .............
            .............  INTERRUPT MASKS
            .............
00000001    EXT_BUS        EQU  1          .PRIORITY 0, EXTERNAL BUS PARITY
                                            ERROR
00000002    IO_TIMEOUT     EQU  2          .PRIORITY 1, I/O TIME OUT
00000004    Data memory_PE      EQU  4          .PRIORITY 2, Data memory MEMORY
                                                 PARITY
                                            ERROR
00000008    ATTENTION      EQU  8          .PRIORITY 3, SUPPORT SYSTEM
                                            ATTENTION
00000010    CP_EXCP        EQU  X'10'      .PRIORITY 4, CP EXCEPTION
00000020    CP_COMP        EQU  X'20'      .PRIORITY 5, CP COMPLETION
            .............
00000008    INTR_MASK      EQU  ATTENTION .INTERRUPT MASK
            .............
            .............  META/PC INTERFACE OPERATIONS
            .............
00000020    META_CNTL      EQU  META_PC+0 .WRITE CONTROL
00000020    META_WRITE     EQU  META_PC+1 .WRITE DATA
00000020    META_STATUS    EQU  META_PC+0 .READ STATUS
00000021    META_READ      EQU  META_PC+1 .READ DATA
            .............
000000C8    MMX_ERR_BS     EQU  200        .MMX IO ERROR MESSAGE BIAS IN
                                            VF77.MTX
            .............  PC/META STATUS CODES
            .............
```

CODE STATEMENT

```
.............
............. EXTENDED MEMORY CHANNEL ASSIGNMENTS
.............
............. CHANNEL 0 = CODE-BODY CURRENT WORD + 1, WORD MODE
............. CHANNEL 1,2 = SCRATCH CHANNELS, NO POST USAGE
              ASSUMPTIONS
............. CHANNEL 3   = SCRATCH, BUT ASSUMED TO BE LEFT IN
              WORD MODE
............. WITH STRIDE = 1 AND TIPTOE = 0
.............
............. META/EXTENDED MEMORY OPERATIONS
.............
............. 'SET'     OPERATIONS-CONTROLLER INITIALIZATION
............. 'LOAD'    OPERATIONS-ESTABLISH LOCATION COUNTER
                        VALUES
............. 'READ'    OPERATIONS-MEMORY TO PROCESSOR TRANSFERS
............. 'WRITE'   OPERATIONS-PROCESSOR TO MEMORY TRANSFERS
............. 'STATUS'  OPERATIONS-CONTROLLER TO PROCESSOR
                        TRANSFERS
.............
............. 'SET' CONTROL WORD FORMAT
.............
............. 31----------------9 8--------4    3       2    1------0
.............   STRIDE, 2's COMP.  TIP TOE  UP/DOWN ORDER  FORMAT

.............
............. 'LOAD' WORD FORMAT
.............
............. 31---24 23------------------------0
.............   RFU      BYTE, HLFWRD OR WRD ADDR
.............
............. RFU = RESERVED FOR FUTURE USE, CURRENTLY RECEIVED AS
              ZEROS
.............
............. 'STATUS' WORD FORMATS
.............
............. (STATUS 1)
.............
............. 31----24 23----------------0
.............   ZERO     CURRENT ADDRESS
.............
............. (STATUS 2)
.............
............. 31  30  29  28  27  26  25  24  23 22  21  20 19  18
              17  16
............. WRP WRA   0   0 BE1 BE0 CP1 CP0 ILV CYC EMT   0 IDX
              WRT CH1 CH0
.............
```

```
             .............  15  14  13  12  11  10   9   8   7   6   5
                            4----------------0
             .............  CHG RFG ERY MST BB3 BB2 BB1 BB0 VAL RPD BY1 RESIDUAL
                            TIP TOE
             .............
             .............  CONTROLLER STATUS
             .............
             .............          WRP = THREE WRITES PENDING
             .............          WRA = ANY WRITE PENDING
             .............      BE1,BE0 = 1 BANK(0), 2 BANKS(1), 4 BANKS(3)
             .............      CP1,CP0 = 256K  RAMS(0),  1  MEGABIT  RAMS(1),  4
                                          MEGABIT RAMS(2)
             .............          ILV = INTERLEAVE ENABLED
             .............          CYC = MEMORY CYCLE IN PROGRESS
             .............          EMT = FIFO IS EMPTY
             .............          IDX = INDEX THIS CYCLE
             .............          WRT = WRITE CYCLE
             .............      CH1,CH0 = CHANNEL FOR CURRENT CYCLE
             .............          CHG = CHANNEL CYCLE,
             .............          RFG = REFRESH CYCLE
             .............          ERY = EARLY REFRESH
             .............          MST = MUST REFRESH
             .............      BB(3-0) = BANK 3,2,1,0 BUSY
             .............
             .............  CHANNEL SPECIFIC STATUS
             .............
             .............          VAL = Data memory VALID
             .............          RPD = READ PENDING
             .............          BY1 = INDEX BY ONE
             .............
             .............  GENERIC MEMORY OPERATIONS
             .............
00000000     MEM_SET        EQU  0          .GENERIC SET
00000001     MEM_WRT        EQU  1          .GENERIC WRITE
00000002     MEM_WRTI       EQU  2          .GENERIC WRITE AND INDEX
00000004     MEM_LD         EQU  4          .GENERIC LOAD
00000005     MEM_LDF        EQU  5          .GENERIC LOAD AND FETCH
00000006     MEM_LDFI       EQU  6          .GENERIC LOAD, FETCH AND INDEX
00000007     MEM_F          EQU  7          .GENERIC FETCH
             .............
00000000     MEM_STAT1      EQU  0          .GENERIC STATUS 1 READ
00000001     MEM_STAT2      EQU  1          .GENERIC STATUS 2 READ
00000003     MEM_RD         EQU  3          .GENERIC READ
00000004     MEM_RDF        EQU  4          .GENERIC READ AND FETCH
00000005     MEM_RDFI       EQU  5          .GENERIC READ, FETCH AND INDEX
             .............
             .............  MEMORY INTERFACE VALUES
             .............
00000000     MEM_CH0        EQU  0          .CHANNEL 0 BASE ADDRESS
00000008     MEM_CH1        EQU  X'8'       .CHANNEL 1 BASE ADDRESS
```

```
00000010   MEM_CH2        EQU  X'10'     .CHANNEL 2 BASE ADDRESS
00000018   MEM_CH3        EQU  X'18'     .CHANNEL 3 BASE ADDRESS
           .............
           .............  MEMORY 'SET' SUB-OPERATIONS
           .............
00000000   MEM_WD_MD      EQU  0         .WORD MODE FORMAT (32 BIT)
00000001   MEM_BYTE_MD    EQU  1         .BYTE MODE FORMAT (8 BIT FROM 32 BIT
                                          WORD)
00000002   MEM_HWD_MD     EQU  2         .HALF WORD MODE FORMAT (16 BIT FROM
                                          32 BIT WORD)
00000003   MEM_HBT_MD     EQU  3         .BYTE MODE FORMAT (8 BIT FROM 16
                                          BIT WORD)
           .............
00000000   MEM_HL         EQU  0         .HIGH TO LOW BYTE OR HALFWORD ORDER
00000004   MEM_LH         EQU  4         .LOW TO HIGH BYTE OR HALFWORD ORDER
           .............
00000000   MEM_UP         EQU  0         .INDEX ASCENDING (UP)
00000008   MEM_DOWN       EQU  8         .INDEX DESCENDING (DOWN)
           .............
00000009   MEM_STRD       EQU  9         .STRIDE OFFSET
00000004   MEM_TT         EQU  4         .TIP TOE OFFSET
           .............
           .............  CHANNEL 0 MEMORY OPERATIONS
           .............
00000000   MEM_SET_0      EQU MEM_SET + MEM_CH0     .CHANNEL 0 SET
00000001   MEM_WRT_0      EQU MEM_WRT + MEM_CH0     .CHANNEL 0 WRITE
00000002   MEM_WRTI_0     EQU MEM_WRTI + MEM_CH0    .CHANNEL 0 WRITE AND INDEX
00000004   MEM_LD_0       EQU MEM_LD + MEM_CH0      .CHANNEL 0 LOAD
00000005   MEM_LDF_0      EQU MEM_LDF + MEM_CH0     .CHANNEL 0 LOAD AND FETCH
00000006   MEM_LDFI_0     EQU MEM_LDFI + MEM_CH0    .CHANNEL 0 LOAD, FETCH AND
                                                     INDEX
00000007   MEM_F_0        EQU MEM_F + MEM_CH0       .CHANNEL 0 FETCH
           .............
00000000   MEM_STAT1_0    EQU MEM_STAT1 + MEM_CH0   .CHANNEL 0 STATUS 1 READ
00000001   MEM_STAT2_0    EQU MEM_STAT2 + MEM_CH0   .CHANNEL 0 STATUS 2 READ
00000003   MEM_RD_0       EQU MEM_RD + MEM_CH0      .CHANNEL 0 READ
00000004   MEM_RDF_0      EQU MEM_RDF + MEM_CH0     .CHANNEL 0 READ AND FETCH
00000005   MEM_RDFI_0     EQU MEM_RDFI + MEM_CH0    .CHANNEL 0 READ, FETCH AND
                                                     INDEX
           .............
           .............  CHANNEL 1 MEMORY OPERATIONS
           .............
00000008   MEM_SET_1      EQU     MEM_SET+MEM_CH1   .CHANNEL 1 SET
00000009   MEM_WRT_1      EQU     MEM_WRT+MEM_CH1   .CHANNEL 1 WRITE
0000000A   MEM_WRTI_1     EQU     MEM_WRTI+MEM_CH1  .CHANNEL 1 WRITE AND INDEX
0000000C   MEM_LD_1       EQU     MEM_LD+MEM_CH1    .CHANNEL 1 LOAD
0000000D   MEM_LDF_1      EQU     MEM_LDF+MEM_CH1   .CHANNEL 1 LOAD AND FETCH
0000000E   MEM_LDFI_1     EQU     MEM_LDFI+MEM_CH1  .CHANNEL 1 LOAD, FETCH AND
                                                     INDEX
0000000F   MEM_F_1        EQU     MEM_F+MEM_CH1     .CHANNEL 1 FETCH
```

```
            ..............
00000008    MEM_STAT1_1    EQU   MEM_STAT1+MEM_CH1 .CHANNEL 1 STATUS 1 READ
00000009    MEM_STAT2_1    EQU   MEM_STAT2+MEM_CH1 .CHANNEL 1 STATUS 2 READ
0000000B    MEM_RD_1       EQU   MEM_RD+MEM_CH1    .CHANNEL 1 READ
0000000C    MEM_RDF_1      EQU   MEM_RDF+MEM_CH1   .CHANNEL 1 READ AND FETCH
0000000D    MEM_RDFI_1     EQU   MEM_RDFI+MEM_CH1  .CHANNEL 1 READ, FETCH AND
                                                    INDEX

            ..............
            .............. CHANNEL 2 MEMORY OPERATIONS
            ..............
00000010    MEM_SET_2      EQU   MEM_SET+MEM_CH2   .CHANNEL 2 SET
00000011    MEM_WRT_2      EQU   MEM_WRT+MEM_CH2   .CHANNEL 2 WRITE
00000012    MEM_WRTI_2     EQU   MEM_WRTI+MEM_CH2  .CHANNEL 2 WRITE AND INDEX
00000014    MEM_LD_2       EQU   MEM_LD + MEM_CH2  .CHANNEL 2 LOAD
00000015    MEM_LDF_2      EQU   MEM_LDF + MEM_CH2 .CHANNEL 2 LOAD AND FETCH
00000016    MEM_LDFI_2     EQU   MEM_LDFI+MEM_CH2  .CHANNEL 2 LOAD, FETCH AND
                                                    INDEX
00000017    MEM_F_2        EQU   MEM_F + MEM_CH2   .CHANNEL 2 FETCH
            ..............
00000010    MEM_STAT1_2    EQU   MEM_STAT1+MEM_CH2 .CHANNEL 2 STATUS 1 READ
00000011    MEM_STAT2_2    EQU   MEM_STAT2+MEM_CH2 .CHANNEL 2 STATUS 2 READ
00000013    MEM_RD_2       EQU   MEM_RD + MEM_CH2  .CHANNEL 2 READ
00000014    MEM_RDF_2      EQU   MEM_RDF + MEM_CH2 .CHANNEL 2 READ AND FETCH
00000015    MEM_RDFI_2     EQU   MEM_RDFI+MEM_CH2  .CHANNEL 2 READ, FETCH AND
                                                    INDEX

            ..............
            .............. CHANNEL 3 MEMORY OPERATIONS
            ..............
00000018    MEM_SET_3      EQU   MEM_SET+MEM_CH3   .CHANNEL 3 SET
00000019    MEM_WRT_3      EQU   MEM_WRT+MEM_CH3   .CHANNEL 3 WRITE
0000001A    MEM_WRTI_3     EQU   MEM_WRTI+MEM_CH3  .CHANNEL 3 WRITE AND INDEX
0000001C    MEM_LD_3       EQU   MEM_LD + MEM_CH3  .CHANNEL 3 LOAD
0000001D    MEM_LDF_3      EQU   MEM_LDF + MEM_CH3 .CHANNEL 3 LOAD AND FETCH
0000001E    MEM_LDFI_3     EQU   MEM_LDFI+MEM_CH3  .CHANNEL 3 LOAD, FETCH AND
                                                    INDEX
0000001F    MEM_F_3        EQU   MEM_F + MEM_CH3   .CHANNEL 3 FETCH
            ..............
00000018    MEM_STAT1_3    EQU   MEM_STAT1+MEM_CH3 .CHANNEL 3 STATUS 1 READ
00000019    MEM_STAT2_3    EQU   MEM_STAT2+MEM_CH3 .CHANNEL 3 STATUS 2 READ
0000001B    MEM_RD_3       EQU   MEM_RD+MEM_CH3    .CHANNEL 3 READ
0000001C    MEM_RDF_3      EQU   MEM_RDF+MEM_CH3   .CHANNEL 3 READ AND FETCH
0000001D    MEM_RDFI_3     EQU   MEM_RDFI+MEM_CH3  .CHANNEL 3 READ, FETCH AND
                                                    INDEX
```

```
CODE        STATEMENT
            .............
            .............  MATH COPROCESSOR OPERATIONS
            .............
            .............  'FUNCTION' OPERATIONS - ESTABLISH FUNCTION AND
                           TRANSMIT FIRST ARGUMENT
            .............  'WRITE' OPERATIONS - PROCESSOR TO COPROCESSOR
                           TRANSFERS
            .............  'READ' OPERATIONS - COPROCESSOR TO PROCESSOR
                           TRANSFERS
            .............
            .............  'FUNCTION' WORD FORMAT (EXCEPT INITIATE)
            .............
            .............  31--------0
            .............    ARGUMENT
            .............
            .............  'FUNCTION' WORD FORMAT (INITIATE)
            .............
            .............  31--16  15  14   13   12  11-------0
            .............    NU   RUN LOAD DUMP STEP   ADDRESS
            .............
            .............  'STATUS' WORD FORMAT
            .............                ____ALU_____  _________MLU_________
            .............  31 30 29  28  29--23 22-------16 15--10  9  8 7  6
                           5  4  3  2  1  0
            .............  RUN WI WO HALT  NU    ADDRESS    NU   OVR UND INX
                           INV DEN OVR UND INX INV RCO
            .............
            .............  RUN = RUNNING
            .............  WI = WAITING FOR INPUT
            .............  WO = WAITING FOR OUTPUT
            .............  HALT = HALTED
            .............  OVR = OVERFLOW
            .............  UND = UNDERFLOW
            .............  INX = INEXACT
            .............  INV = INVALID
            .............  DEN = DENORMALIZED
            .............  RCO = ROUND CARRY OUT
            .............
000003FF    CP_EXCPMSK     EQU  X'3FF'        . MASK FOR ALL EXCEPTION BITS
00000001    CP_MLU_RCO     EQU  1             . MLU ROUND CARRY OUT MASK
00000002    CP_MLU_INV     EQU  2             . MLU INVALID MASK
00000004    CP_MLU_INX     EQU  4             . MLU INEXACT MASK
00000008    CP_MLU_UND     EQU  8             . MLU UNDERFLOW MASK
00000010    CP_MLU_OVR     EQU  X'10'         . MLU OVERFLOW MASK
00000020    CP_MLU_DEN     EQU  X'20'         . MLU DENORMALIZED MASK
00000040    CP_ALU_INV     EQU  X'40'         . ALU INVALID MASK
00000080    CP_ALU_INX     EQU  X'80'         . ALU INEXACT MASK
00000100    CP_ALU_UND     EQU  X'100'        . ALU UNDERFLOW MASK
00000200    CP_ALU_OVR     EQU  X'200'        . ALU OVERLFOW MASK
```

```
                .............
                .............  COPROCESSOR 'FUNCTION' OPERATIONS (EXCEPT IF 'LOAD'
                               OR 'DUMP' ACTIVE)
                .............
00000038        CP_ADDR        EQU  X'38'         . COPROCESSOR BASE ADDRESS
                .............
00000038        CP_INTMPY      EQU  CP_ADDR+0     . INTEGER*32 MULTIPLY (I*J),
                                                    TRANSMIT 'I'
00000039        CP_FPADD       EQU  CP_ADDR+1     . REAL*32 ADDITION (A+B),
                                                    TRANSMIT 'A'
0000003A        CP_FPSUB       EQU  CP_ADDR+2     . REAL*32 SUBTRACTION (A-B),
                                                    TRANSMIT 'A'
0000003B        CP_FPMPY       EQU  CP_ADDR+3     . REAL*32 MULTIPLY (A*B),
                                                    TRANSMIT 'A'
0000003C        CP_FPDADD      EQU  CP_ADDR+4     . REAL*64 ADD (A1,A2+B1,B2),
                                                    TRANSMIT 'A1'
0000003D        CP_FPDSUB      EQU  CP_ADDR+5     . REAL*64 SUB, (A1,A2-B1,B2),
                                                    TRANSMIT 'A1'
0000003E        CP_FPDMPY      EQU  CP_ADDR+6     . REAL*64 MULT,(A1,A2*B1,B2),
                                                    TRANSMIT 'A1'
0000003F        CP_INIT        EQU  CP_ADDR+7     . INITIATE OPERATION (X),
                                                    TRANSMIT 'X'
                .............
00008000        CP_RUN         EQU  X'8000'       . 'RUN' BIT (NORMAL EXECUTION
                                                    MODE)
00004000        CP_LOAD        EQU  X'4000'       . 'LOAD' BIT (Interpreter WRITE
                                                    MODE)
00002000        CP_DUMP        EQU  X'2000'       . 'DUMP' BIT (Interpreter READ
                                                    MODE)
00001000        CP_STP         EQU  X'1000'       . 'STEP' BIT (SINGLE STEP MODE)
                .............
                .............  INITIATE OPERATION - ADDRESS CROSS-REFERENCE
                .............
00000008        CP_HALTADD     EQU  8             . ADDRESS TO SETUP FOR NEXT
                                                    FUNCTION
00000000        CP_INTDIV      EQU  0             . INTEGER DIVISION (INVALID)
000000E0        CP_I2R         EQU  X'E0'         . INTEGER TO REAL*32 CONVERSION
000000F0        CP_R2I         EQU  X'F0'         . REAL*32 TO INTEGER CONVERSION
00000100        CP_I2RD        EQU  X'100'        . INTEGER TO REAL*64 CONVERSION
00000110        CP_RD2I        EQU  X'110'        . REAL*64 TO INTEGER CONVERSION
00000120        CP_R2RD        EQU  X'120'        . REAL*32 TO REAL*64 CONVERSION
00000130        CP_RD2R        EQU  X'130'        . REAL*64 TO REAL*32 CONVERSION
00000140        CP_FPDIV       EQU  X'140'        . REAL*32 DIVISION
00000170        CP_FPDDIV      EQU  X'170'        . REAL*64 DIVISION
                .............
                .............  COPROCESSOR 'FUNCTION' OPERATIONS (WHEN 'LOAD'
                               ACTIVE)
                .............
00000038        CP_HDATA       EQU  CP_ADDR+0     . TRANSMIT MOST SIGNIFICANT
                                                    MICRO-WORD (47-32)
```

```
00000039    CP_LDATA        EQU  CP_ADDR+1      . TRANSMIT LEAST SIGNIFICANT
                                                  MICRO-WORD (31-0)
            .............
            .............   COPROCESSOR 'WRITE' OPERATIONS DURING 'FUNCTION'
                            EXECUTION
            .............
00000038    CP_OPER         EQU  CP_ADDR+0      . WRITE NEXT OPERAND, TRANSMIT
                                                  OPERAND
00000039    CP_STEP         EQU  CP_ADDR+1      . STEP COPROCESSOR (STEP MODE
                                                  ONLY)
0000003A    CP_RESET        EQU  CP_ADDR+2      . TERMINATE FUNCTION AND RESET
                                                  OPERATION
            .............
            .............   COPROCESSOR 'READ' OPERATIONS DURING 'FUNCTION'
                            EXECUTION
            .............
00000038    CP_RSLT         EQU  CP_ADDR+0      . ARITHMETIC RESULT*32
00000039    CP_BUS          EQU  CP_ADDR+1      . COPROCESSOR INTERNAL BUS*32
0000003A    CP_STAT         EQU  CP_ADDR+ 2     . STATUS
            .............
            .............   COPROCESSOR 'READ' OPERATIONS (WHEN 'DUMP' ACTIVE)
            .............
00000038    CP_RHDATA       EQU  CP_ADDR+0      . RECEIVE MOST SIGNIFICANT
                                                  MICRO-WORD (47-32)
00000039    CP_RLDATA       EQU  CP_ADDR+1      . RECEIVE LEAST SIGNIFICANT
                                                  MICRO-WORD (31-0)
```

```
CODE        STATEMENT
            .............
            ............. PCAT INTERFACE
            .............
00000020    PCAT_ADDR      EQU  X'20'
            ............. READ ADDRESSES
00000020    PCAT_RSTAT     EQU  PCAT_ADDR+0
00000020    PCAT_WCTRL     EQU  PCAT_ADDR+0
00000021    PCAT_WDATA     EQU  PCAT_ADDR+1
            .............
            ............. STATUS BITS
            .............
00000100    PCST_DATATO    EQU  X'100'        . DATA WORD AVAILABLE TO PCAT
00000200    PCST_CTRLTO    EQU  X'200'        . CONTROL WORD AVAILABLE TO PCAT
00000400    PCST_PARTO     EQU  X'400'        . VALID PARITY TO PCAT
00000001    PCST_DATAFR    EQU  X'1'          . DATA WORD FROM PCAT
00000002    PCST_CTRLFR    EQU  X'2'          . CTRL WORD FROM PCAT
            .............
            ............. ERROR MESSAGES
            .............
00000064    MMIOMSGBIAS    EQU  100
            .............
00000037    ERR_BAD_ST     EQU  55            . ERROR IN STATUS PROTOCOL
00000038    ERR_NO_DTA     EQU  56            . MISSING DTA WORD
00000039    ERR_NO_EOM     EQU  57            . MISSING EOM WORD
            .............
            ............. ERROR MESSAGE #S
            .............
00000029    RD_EOF         EQU  41            . END OF FILE ON READ
00000050    MSG_NOPEN      EQU  80            . MESSAGE FILE DID NOT OPEN
            .............
00000002    CMD_CL         EQU  2             . CLOSE
00000008    CMD_OP         EQU  8             . OPEN
00000004    CMD_WM         EQU  4             . WRITE MESSAGE
00000009    CMD_RD         EQU  9             . READ
0000000C    CMD_ST         EQU  12            . STOP
0000000E    CMD_WR         EQU  14            . WRITE
00000010    CMD_LD         EQU  16            . LOAD
00000001    SCMD_PRG       EQU  1             . LOAD PROGRAM
00000013    CMD_TD         EQU  19            . TIME-DATE
0000001E    CMD_RESET      EQU  30            . RESET
0000001F    CMD_EOM        EQU  31            . END OF MESSAGE
            .............
00000001    EXEC_CMD       EQU  1             . EXECUTE (IGNORED)
00000002    HALT_CMD       EQU  2             . HALT (IGNORED)
00000003    ACK_CMD        EQU  3             . ACKNOWLEDGE
00000004    STAT_CMD       EQU  4             . STATUS FOLLOWS
00000005    DTA_CMD        EQU  5             . DATA FOLLOWS
00000006    ATTN_CMD       EQU  6             . ATTENTION (IGNORED)
0000001E    RESET_CMD      EQU  30            . RESET
```

```
0000001F    EOM_CMD         EQU  31             . END OF MESSAGE
            .............
00004000    FORMATTED       EQU  X'4000'
00008000    UNFORMATTED     EQU  X'8000'
```

INDEX

"A" . . . 37
"A" field . . . 34, 36
"B" . . . 37
"B" field . . . 34, 39, 40, 42
"C" field . . . 42, 45
"K Register" . . . 30
"LINK" . . . 31
"op" field . . . 37
"pipeline" . . . 43
"RETURN" . . . 31
"T" field . . . 34, 35
"von-Neumann bottleneck" . . . 3
(<T>:=<A>op <B>) . . . 48
(<T>:=<A>op<B>) . . . 43
<A> . . . 26, 30
<B> . . . 26, 30
<T> . . . 26
<T> := <A> op <B> . . . 26
<T>:=<A>op<B> . . . 34
#DEFINE . . . 100
#IF #ELSE #END . . . 100
#INCLUDE . . . 100
Address field . . . 42, 43, 45, 47
Assembler . . . 1
Assembly formats . . . 1
Backus-Naur Form . . . 1
BLOCKS, NOBLOCKS (BLOCK STRUCTURE ANNOTATIO . . . 97
BNF . . . 1
BUS . . . 50
BUS EMIT . . . 29
BUS REC . . . 29
Caching . . . 4
CHANGE, UNCHANGE (KEYWORD ALTERATION) . . . 95
CISC . . . 26
Clock cycle . . . 26, 45, 48
CODELEN . . . 99
Complex-instruction-set computer (CISC) . . . 3
Composite instruction . . . 24, 26, 28, 29
COMWIDTH (COMMENT WIDTH FOR JUSTIFICATION) . . . 98
Conditional transfer . . . 43, 48
Conditional transfer instructions . . . 43
DATA . . . 27
Data Register (DR) . . . 31
DC (DEFINE CONSTANT) . . . 96
Decoding . . . 43
EB . . . 32
EIR . . . 32, 33

EIR External Input Register 27
EJECT (PAGE EJECTION) 97
Encoding 3, 34
END (END OF ASSEMBLY 98
EQU (EQUATE SYMBOLS) 96
External bus 7, 24, 28, 29, 32
External input register 24, 32
External input register (EIR) 29
FIGURE 1 5
FIGURE 2 10
FIGURE 3 11
FIGURE 4 12
FIGURE 5 18
FIGURE 6 25
FIGURE 7 . . . . 67, 75
FIGURE 8 68
FIGURE 9 71
Flush 44
FORMAT, NOFORMAT (FORMATTED LISTING CONTROL) 98
HEADER (PAGE HEADINGS) 97
IBM PC-AT 2
IFON, IFOFF 99
ILIST, NOLIST (INCLUDE FILE PRINTING ON, OFF 98
IMR 32, 47
IMR Interrup Mask Register 27
INST, DATA (INSTRUCTION, DATA MODE INITIATION) 95
Instruction Pipeline 26
Instruction set architecture 6
Interrupt 24, 28
Interrupt Mask Register 47
Interrupt Mask Register (IMR) 32
Interrupt Register 47
Interrupt Register (IR) 32
IOTMO (input/output timeout 33
IR 47
IR Interrupt Register 27
K 29, 30
K Register 31, 45
KR K Register 27
LABEL 94
Language interpretation 3, 4, 7
LDK 45
Left Hand Side Format 26
LHS 24, 26, 29, 34
Link 44, 46
Link Conditional 46
Link Stack Register (LSR) File 31
LINKC 46
LIST, NOLIST (ASSEMBLY LISTING ON, OFF) 97
LISTC, NOLISTC (LISTING COMMANDS ON, OFF 97

Logical shift 41
LSR Link Stack Register File 27
MAR 29, 31, 45
MAR Memory Address Register 27
Memory Address Register (MAR) 30
Memory Input Register (MIR) 30
Memory Output Register (MOR) 31
Meta architecture 3
Metacomputer 6
MetaMicro 7, 29, 31, 32, 34, 43, 45
MnetaMicro Processor 6
User memory 6, 29
METASYSTEM 6, 7
Data memory 6, 7, 29, 30, 31, 45
Instruction set 1
Instruction memory 6, 7, 43, 45
Microsystem: Minimal Instruction Set Computer (MISC) . . 6
Minimal-instruction-set-computer (MISC) 3
MIR 29, 30
MIR Memory Input Register 27
MOR 29, 31
MOR Memory Output Register 27
NOP 38
Off-Chip Interfaces 29
Op 26
OP FIELD 38
ORG (SET ORIGIN OF ABSOLUTE CODE) 95
Overflow 41, 42
PC-Channel 6, 29
Pipeline 45
Priority bit 47
R0 ... R7 General Registers 27
Reduced-instruction set computer (RISC) 3
Return 44, 47
RHS 24, 34, 44
RHS external bus 29
Right Hand Side Format 28
RISC 26
Rotational 42
Rotational shift 41
RS (RESERVE STORAGE) 96
Shift 40, 42
Shift operations 40
Shifting 40
Shifts 41
SKIP 48
Skip Conditional 48
Skips 28
Subroutine link/return 28
Subroutine linkage 47

Subroutine return 46
TRA 43
Transfer 44
Transfer address 47
Transfers 28
Unconditional transfer 45
VLSI feature size 3
Von-Neumann bottleneck 3
WARN, NOWARN 98
X Result Register 27
X Register 30