Report No. 4268

MBB Microprogrammer's Handbook

R. Weissler, M. Kraley, P. Herman

First Published January 1980

Revised August 1980

Table of Contents

PICTURES

1 INTRODUCTION AND OVERVIEW

The Microprogrammable Building Block (MBB) is a general purpose, easily microprogrammable, powerful computer which can be used for a variety of applications. This manual attempts to completely describe the MBB's hardware elements and their function from the microprogrammer's point of view.

Although the MBB can be programmed directly in its microlanguage for a specific application we envision that in most application, the MBB will be emulating some other machine, either existing or invented. Accordingly, we begin with a discussion of emulation, which also introduces some terms which will be used throughout this document. We then describe some of the physical characteristics of the machine and give an overview of its logical structure.

1.1 MPMs and Emulation

Because they allow close program control over hardware action, microprogrammable computers are often suited for emulating other computers. When used for such emulation, the microprogrammable machine (MPM) is called the "micromachine" or "microcomputer"; its code is "microcode", its instructions are "microinstructions", the execution of a microinstruction is a "microcycle", and so on. (We are not using "microcomputer" to mean a very small computer, as is sometimes done.) The emulated machine is called the "macromachine", its code is "macrocode", and so on.

To emulate the macrocomputer's execution of a given macroprogram, the microcomputer needs that program loaded as data. The microcomputer's microprogram reads macroinstructions one at a time, decoding each one and emulating its execution. For example, a macroprogram branch corresponds to the microprogram's breaking out of sequence in its examination of macroinstructions.

Emulating a macroinstruction is often straightforward. Macroinstructions generally deal with macroregisters, macromemory, and macro-I/O (including interrupts); and the macromachine's data operations generally consist of unary and binary operations such as NOT and ADD. The microcomputer generally contains registers which the microprogram uses as images of macroregisters. For example, one of the microregisters is maintained to have its contents reflect the macromachine's program counter, and is used to determine the next macroinstruction to be emulated. Normally, emulating the execution of a macroinstruction would involve increasing that register by one; emulating the execution of a macro-JUMP command would involve setting that register to reflect the new macromachine's execution address. Other microregisters would mirror the macromachine's accumulators and index registers. A macroinstruction to clear an accumulator, for example, would be emulated by clearing the corresponding microregister.

The microcomputer also generally has a memory area which the microprogram uses as a direct image for macromemory. This area holds the macromachine's program and data. To fetch the next macroinstruction to be executed, the microprogram simply reads from this area at an address indicated by its image of the macromachine's program counter. To emulate a macroinstruction to clear a memory cell, the microprogram simply clears the corresponding location in its image of macromemory.

Further, the microcomputer has, in general, an Arithmetic and Logic Unit (ALU) which can perform the macrocomputer's data operations such as NOT and ADD. To simulate a macroinstruction to add memory contents to an accumulator, the microprogram uses its ALU to add the contents of its image of the memory cell to the contents of its image of the accumulator and stores the result in its image of the accumulator.

Because the macromachine's registers, memory, and data operations have direct representations, emulating their functions is easy. In contrast, the macromachine's I/O system often is not replicated. For example, the macrocomputer may have many priority-ordered interrupt levels, while the microcomputer may make available only one; the macrocomputer's I/O instructions may send signals down an I/O bus which has no equivalent in the microcomputer; or the macrocomputer may have sophisticated hardware which lets it initiate a block transfer with one instruction, while the microcomputer lacks such hardware and must transfer each word separately under microcode control. When I/O hardware is not mirrored precisely, emulating I/O functions can be difficult. Generally, the micromachine's system I/O handlers interact with microcode which emulates I/O-related macroinstructions such as instructions to request an interrupt or to initiate a block transfer.

In addition to I/O hardware, certain other specialized hardware may not be mirrored. For example, the micromachine may lack the macromachine's hardware logic for decoding macroinstructions or performing memory mapping. In all of these cases, the microcode assumes the burden of emulating the macromachine's specialized hardware. The microcode's close control over the microcomputer's hardware is usually important for efficient emulation.

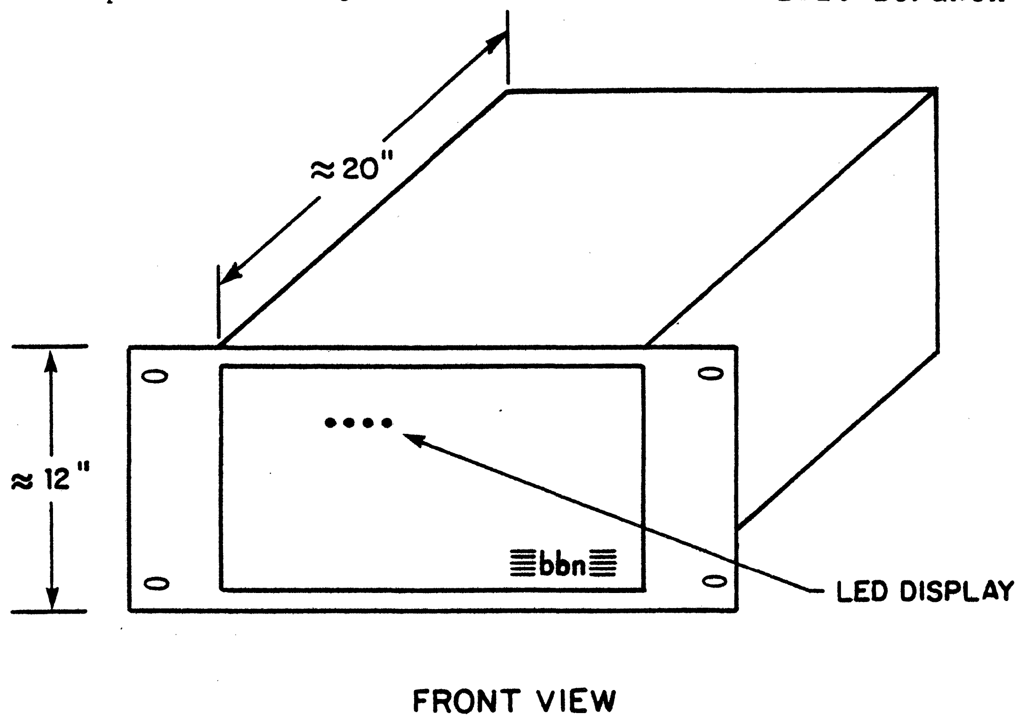
An MPM used for emulation may offer two advantages over the macrocomputer it emulates. First, the MPM may be cheaper, since it can use simple, inexpensive, regularly structured hardware, using microcode to mimic the macromachine's expensive specialized hardware features. Second, it may be easy to change the microcode to enhance the macrocomputer's configuration, primitive operation, or efficiency. For example, it might be easy to expand macromemory, to add new macroinstructions, or to optimize the emulation of frequently executed macrocode.

1.2 Physical Description

Before discussing the detailed logical design of the MBB we will describe the MBB's physical structure. Figures 1.1a through 1.1c show various views of a minimum MBB configuration. The exact configuration is dependent on the intended application.

The housing is a box approximately 12" high and 20" deep, mountable in a standard 19" rack. A smoked plexiglass front lets display lights shine through. Inside, four printed circuit (PC) cards rest between guide rails; one is for the power supply, two are for the basic MBB processor/memory system, and one is available for I/O circuitry. (The power supply card has some logic circuitry in addition to fans, power supply, and other bulky units. For convenience it too is called simply a "card".) Because the cards are fairly wide (about 14"), each has a transverse stiffener to prevent bowing.

Each card has a metal bulkhead mounted at the rear. The cards use L-shaped electrical connectors, one end soldered to the card and the other mounted in the bulkhead. All electrical



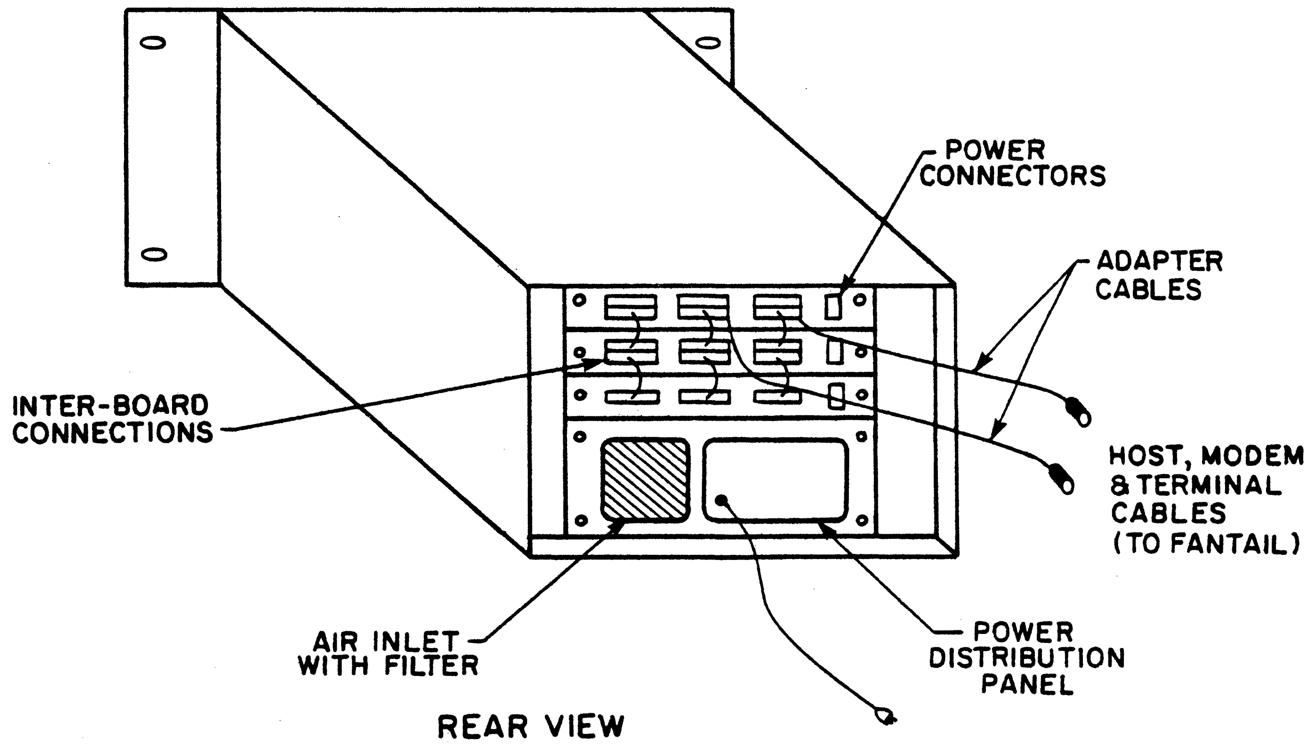


Figure 1.1a: MBB Physical Structure
Picture 1

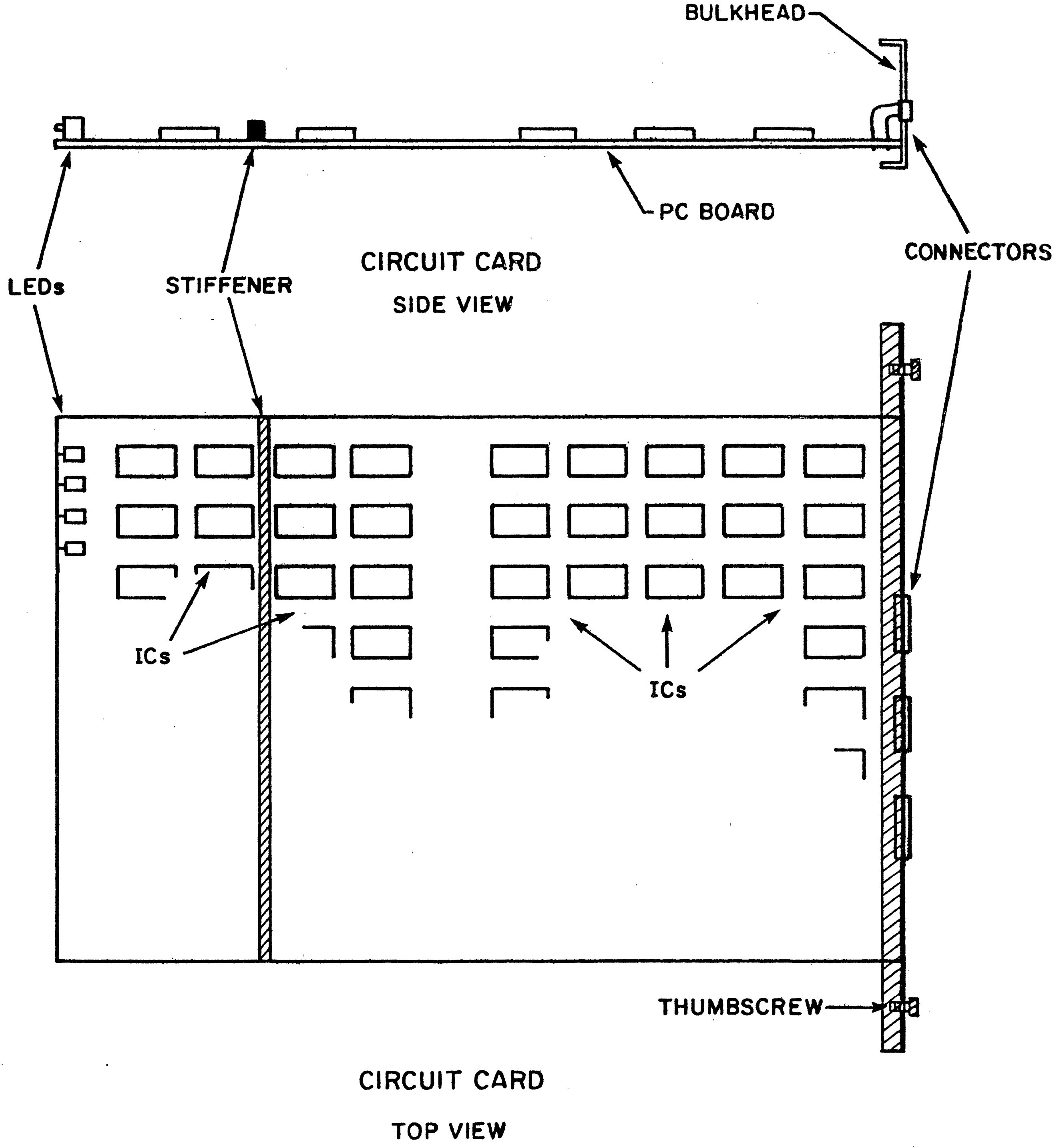


Figure 1.1b: MBB Physical Structure (continued)
Picture 2

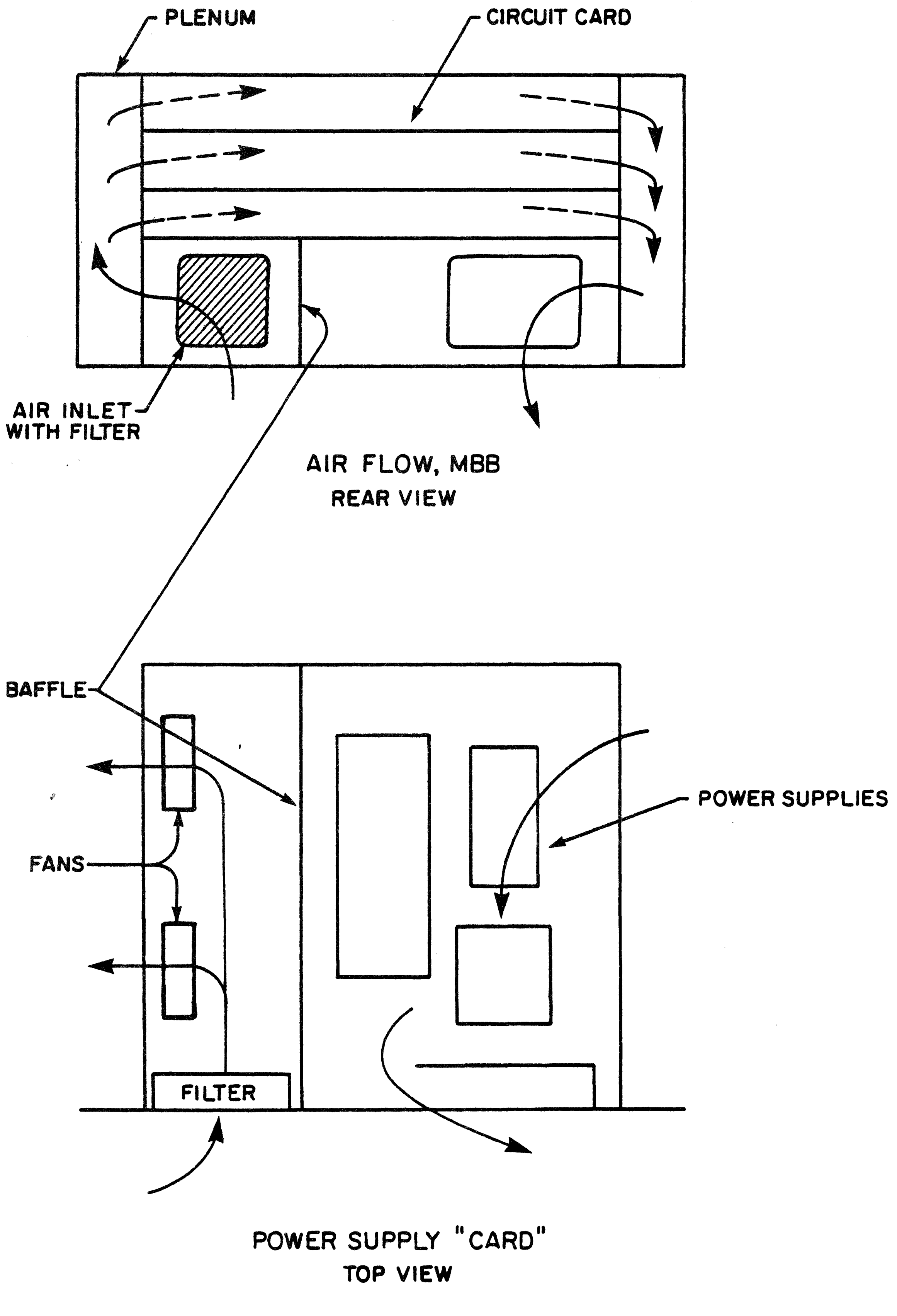


Figure 1.1c: MBB Physical Structure (continued)
Picture 3

connections from one card to another are made with cables which attach to the connectors mounted on the bulkhead. The bulkhead is wider than the card, and it transfers the force of connector mating to the side rails of the box. The bulkhead screws into the box to keep the card still. The bulkhead, like the transverse stiffener mentioned above, helps to prevent bowing.

The MBB's side walls extend past the rear; the overhang protects the connectors mounted on the bulkheads.

Since the box has no internal wiring or edge connectors, tight tolerances are not needed for placement of cards in the box.

PC cards are spaced rather widely to ease cooling, relax mechanical tolerances, and provide sufficient area for connectors. Spacing is not uniform; the power card, containing two fans, needs considerable space while an I/O card may need a wide bulkhead to accommodate many connectors.

Power consumption is only about 350 watts at the AC plug, so two fans easily provide adequate cooling. Figure 1.1c shows air flow viewed from the rear. Air flows in at the bottom left rear. across the left half of the power board, up and across the other boards from left to right, down and across the right half of the power board, and out the bottom right rear. A baffle, reaching from the power board up to the next higher board, prevents "short-circuit" air flow between the inlet and outlet sides. The power board has its logic elements on the left side to be cooled by incoming air, and its heavy heat dissipaters on the right side to be at the end of the airflow.

A battery on the power board provides one minute of backup power. Our experience suggests that such a battery will enable the MBB to withstand a large majority of external power outages. The battery will recharge automatically and will require no regular maintenance; under normal conditions it should last 3-5 years.

The MBB's hardware is divided into a "standard" or "system" part, which is present in every MBB system, and an "application" part, which is peculiar to the particular application. The standard part consists of the housing, the power card, and two PC cards for processor/memory. The processor/memory logic is designed to permit two inserts: small PC cards called the MIR and MAR daughterboards. These daughterboards are custom designed for each MBB application.

1.3 Processor Design

1.3.1 Basic Data Loop

Figure 1.2 shows the logical structure of the MBB's data paths. The ALU is the heart of the machine. Forming a loop with the ALU are the source and destination busses and various registers. This loop is referred to as the "processor", and is shown in more detail in Figure 2.1. The source and destination busses are driven by "tri-state" logic; that is, the hardware selects one source of data at a time to drive them, holding the other possible sources in a neutral state. A typical microinstruction determines the ALU's two inputs, an ALU operation, and destinations for the ALU's output. The ALU's "A" input comes from one of the ALU's scratch pad registers. The ALU's "B" input comes from one of the possible inputs to the source bus. The ALU operation may be ADD, SUBtract (scratch register minus source bus), EXOR, OR, AND, PASS source bus, and NOT source bus. The ALU's output may be written to any one register driven by the destination bus; it may also be written back to the same scratch pad register which supplied the "A" input.

The data paths of the MBB are 20 bits wide. While only 16-bit paths are needed for a straightforward emulation of a 16-bit machine, the only significant price paid for the extra bits is the PC board space consumed by the extra data paths; the processor's internal ICs to populate them cost very little. (In contrast, the extra bits for the ALU scratch registers and main memory are reasonably expensive, and are also easier to optionally omit.) The potential use of the extra 4 bits seems well worth the price. These bits could be used to expand the instruction set and address space of a 16-bit machine. Such an

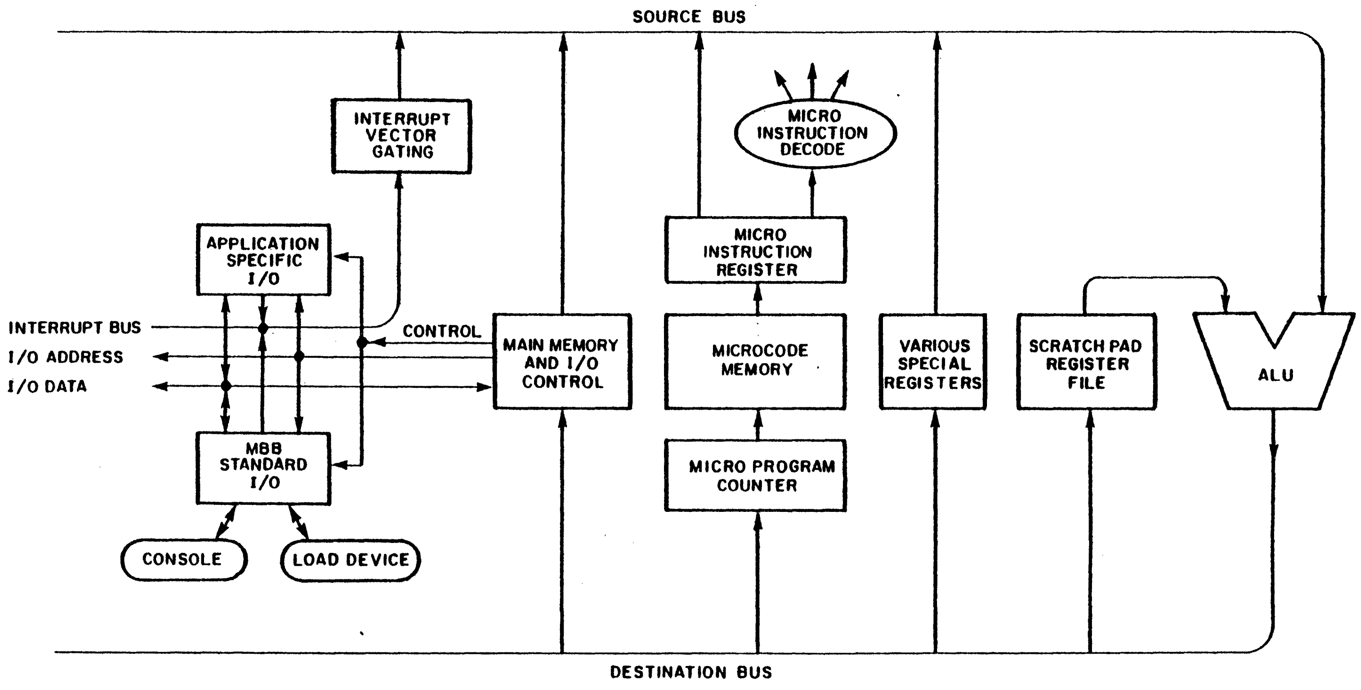


Figure 1.2: MBB Logical Structure
Picture 4

expansion would involve changing the emulation microcode (maintaining compatibility with existing macrocode), and probably could be performed in easy steps.

A software-controlled system status bit specifies whether the machine performs arithmetic operations (such as adding and shifting) and records ALU status based on either 16-bit or 20-bit data words.

The basic microcycle time of a 16-bit MBB is 125 nsec. If an MBB uses the full 20-bit data width, the microcycle time must be increased from 125 nsec to 135 nsec to accommodate a longer ALU carrying propagation time. To effect this change, a different crystal is used to control the system clock signal.

A microprogram counter, the UPC, controls the fetching of microinstructions from microcode memory. The UPC can be loaded from the destination bus to effect a program branch; therefore, the machine's full data computation power may be used to direct the flow of program execution. The microinstruction being executed is held in the microinstruction register (UIR) for decoding.

Microcode is contained in both PROM and RAM. PROM is read-only and non-volatile. RAM is writable, and its contents are lost on power failure. PROM code is intended for such tasks as a system bootstrap or a mini-DDT; code size is kept small, but code speed is not an issue. For RAM code, in contrast, size is less important and speed is more important.

The ALU with its scratch registers, and the microcode memory coupled with instruction fetching and decoding, provide a basic computing capability. Further power is supplied by other major components, which we now describe.

1.3.2 MIR and Dispatch

A macroinstruction to be emulated may be loaded into the macroinstruction register (MIR). The MIR daughterboard (MIRDB), custom designed for each macromachine to be emulated, assists in macroinstruction decoding. Some simple, application-dependent hardware in the MIRDB can free the microcode from some cumbersome, frequently required calculations which would seriously slow the machine. The MIRDB has three principal functions:

1. to ease macroregister selection
2. to facilitate dispatching to microroutines based on the macroinstruction
3. to provide a specific transformation of the macro-instruction

For register selection, bits from the MIR may be used (under microinstruction control) to select the scratch pad register that feeds the ALU's "A" input. This feature is intended for macromachines whose macroinstructions have a field specifying which of several macroregisters to use. These different macroregisters would be mirrored by a block of ALU scratch pad registers, and a macroinstruction's register specification field would automatically cause the MBB to access the corresponding microregister.

The second function of the MIRDB is to assist the microprogram in dispatching to the proper routines for emulating the various macroinstructions. Bits from the microinstruction (UIR), the macroinstruction (MIR), and the system status register (MISC) are combined in an application-dependent manner to produce a 10-bit "dispatch address". This dispatch address is not itself an address of a microcode routine; such a situation would have required a complicated MIRDB, as well as fixed routine addresses. Instead, these 10 bits address a cell in a 1K x 12-bit "dispatch memory"; this cell holds the microcode address to which to branch. A microinstruction may load this microcode address into the UPC to effect the branch. (Because of memory timing constraints, this address is loaded directly onto the destination bus rather than being loaded onto the source bus and passed through the ALU.) Dispatch memory is easily rewritten, so that locations of emulation routines need not be fixed.

1.3.3 MAR and Main Memory

Another major MBB component is the main memory and associated logic. Main memory is intended to include the macroaddress space (containing the macroprogram's code and variables), plus an area reserved for microcode use. In general, main memory is used, as are the ALU scratch registers, for read/write data. Compared with the scratch register memory, main memory is much larger, but slower (as discussed shortly).

The memory address register (MAR) controls main memory access. When a microinstruction loads the MAR, certain microinstruction bits specify whether to initiate a main memory transfer, an I/O transfer, or neither; whether the transfer is a read or a write; and for main memory transfers, whether the reference is "macro" (using an address supplied by an MAR daughterboard, as explained below) or "physical" (using the address in the MAR directly).

Compared with the MBB's internal processing, the main memory access is rather slow; reading from a main memory cell takes three microcycles. The first microcycle is used to load the MAR with the address desired. The next two cycles may be used freely, except that they may not write to the MAR or to the memory buffer register (MBR). At the start of the following cycle, the contents fetched from main memory are available in the MBR. Writing to main memory proceeds similarly. The MBR must be set up beforehand with the data to be written, the MAR is loaded to specify the memory address, and the two cycles following the MAR load may not alter the MAR or the MBR.

The main memory address selected may be passed directly from the MAR to main memory; or it may instead, under microinstruction control, pass through an MAR daughterboard (MARDB) for translation and validity checking. Main memory references which emulate macromemory references will pass through the MARDB to undergo any address translation and validity checking characteristic of the macromachine. An illegal address detected by the MARDB prevents any memory access and sets a status bit. Since valid addresses are properly translated and illegal addresses prevent access, an emulation of a macromachine memory operation cannot access the area reserved for private microcode use. This private area can only be accessed by a "physical" read or write, in which the address is passed directly from the MAR to the memory.

Like the MIRDB, the MARDB is tailored for each application; here again, a simple piece of hardware may greatly speed up emulation by freeing the microcode from some tedious, frequently required calculations.

On some machines, I/O transfer instructions look like memory reference instructions; the hardware traps references to certain addresses as really referring to certain I/O devices. To emulate such a machine, the MARDB would treat such addresses as illegal, preventing access and setting an easily testable status bit. The microcode could then test this bit and branch to an appropriate routine.

I/O devices are read and written much like main memory. The MAR specifies an I/O address, and the MBR serves as a data buffer. I/O transfers differ from main memory transfers in two ways. First, while main memory transfers take three cycles (including the MAR load), I/O transfers take either two or four, depending on the device. Second, while for main memory the intervening cycles after the MAR load may be used rather freely, for I/O these cycles (one or three, depending on the device) must be devoted to supporting the transfer.

1.3.4 Other Features

Figure 2.1 shows certain other MBB features worth mentioning. To facilitate byte operations, two byte-swapped forms of the MBR are available as sources, one for 8-bit bytes, and one for 10-bit bytes. The MBR was chosen because many of the byte operations desired will probably deal with data bytes read from main memory or I/O devices. To support shift operations, the MAR and MBR form a double-length shift register; one microinstruction may shift their contents right or left one bit (end off, 0 shifted in). We decided not to add more shifting hardware to handle multiple-position shifts and different shift types (rotate, arithmetic). Such hardware, while it would have speeded emulation of shifts for some machines, seemed too specialized and thus not worth the cost. The microcode, aided by our simple shift hardware, assumes the burden of emulating the various types of shift instructions.

2 PROCESSOR

Now that we have had an overview of the processor structure, we can examine the pieces of the MBB in more detail (see Figure 2.1).

Bits in registers and on busses are numbered starting from 0, the low order bit. All numbers are decimal, unless otherwise specified.

2.1 Basic Data Flow

Data flow centers around the Arithmetic Logic Unit (ALU). The ALU has two inputs. Its "A" input is from a scratch pad register file (1024 words * 20 bits). Its B input is from a 20-bit "source bus". All drivers of the source bus, plus Dispatch memory, are called "sources". A 20-bit "destination bus" is driven by the ALU's output (or Dispatch memory, see below). All registers which can be driven from the destination bus, except the ALU's scratch pad registers, are called "destinations".

Selecting the ALU's two inputs, performing an arithmetic or logic operation, and storing the result are the main events in a microcycle. A microinstruction specifies a source to drive the B input, a scratch pad register to drive the A input, an ALU operation, and a single destination to receive the output. (If Dispatch memory is the source, it drives the destination bus

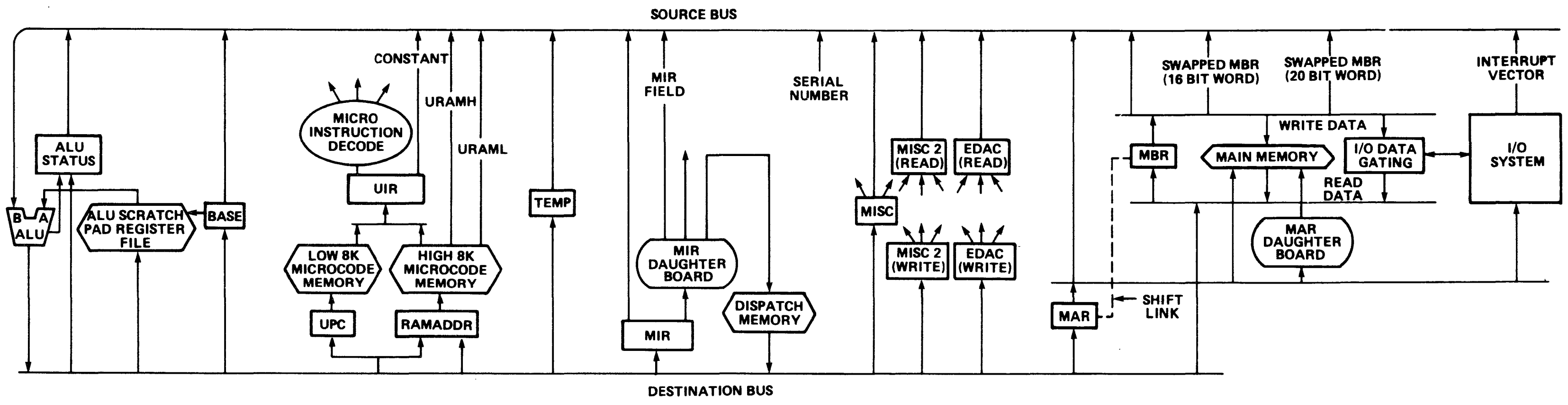


Figure 2.1: Processor
Picture 5

directly, and the ALU is not used.) The ALU's output may optionally be written back to the same scratch pad register which fed the A input.

Some sources are less than 20 bits wide. When they are read onto the source bus, the bus is padded with high order 0's. When registers less than 20 bits wide are written from the destination bus, the high order bits on the bus are ignored.

The seven available ALU operations are:

```
ADD
SUBTRACT  (scratch register minus source bus)
AND
Inclusive OR
Exclusive OR
PASS source bus
NOT source bus (bitwise complement)
```

The eighth possible ALU operation is undefined. The ALU is combinatorial only; it has no storage. To pass a scratch pad register, the assembler uses the ADD operation with a constant of 0 driving the source bus; we may think of passing a scratch register as an available "operation".

2.2 Register Selection

There are 1024 registers; only 32 are accessible at any given time. Sixteen global registers are always accessible; a "window" of 16 registers is movable within the 1K address space depending on the contents of a register called BASE. Registers may also be selected on the basis of the contents of MIR. In detail:

A five-bit microinstruction field selects the scratch pad register for the ALU's input. Let R be the field's value. For R = 0 through 15, the contents of the 10-bit BASE register determines the window of addresses available; BASE is inclusive ORed with R to select the desired register. (This inclusive OR function allows several uses of BASE. If BASE is a multiple of 16, then the local register block is a full sized 16 register block. With the appropriate choice for BASE, the local register block can be half, quarter or eighth sized. If local register 0 is selected, then BASE serves as a direct 10 bit address into the register file.) The set of registers addressable at any moment with R = 0 through 15 is called the current "local register block". Registers 0-13, the "global" registers, are always accessible, using R = 16 through 29. For R = 30 or 31, BASE again sets the window; the particular register in the local block is determined by the Macroinstruction Register daughterboard (MIRDB) in an application-dependent manner. Two different registers may be selected using R = 30 and R = 31. Register addresses supplied by the MIR are also ORed with the contents of BASE, so the registers are always within the current "local" block. The MIR may not be used for register selection on any cycle immediately following a load of the MIR.

In summary, the register address is determined as follows:

<u>value of R</u>	<u>register</u>	
0-15	R!{BASE}	(locals)
16-29	R-16	(globals)
30,31	MIR function}!{BASE}	

If BASE is loaded from the destination bus in cycle N, its new value takes effect for register selection in cycle N+1. The value of BASE when referenced as a source, however, does not change until cycle N+2. This special one cycle delay permits the old value of BASE to be saved in the new local register block.

2.3 ALU Status

Each microinstruction may optionally latch the ALU status. If the microinstruction specifies latching, the following ALU bits and conditions are latched into the ALU Status Register (ALUST):

high bit of A input
high bit of B input
high bit of output
low bit of output
ALU result zero
carry

The "high" bits are bit 19 or 15, depending on whether the machine's data width mode specifies 20 or 16 bits respectively. The "ALU result zero" status bit is set when the ALU result is zero; in 16-bit mode, only the low 16 bits matter. The carry, also dependent on the data width mode, reflects the traditional "carry-out" on addition, and the complement of "borrow-in" on subtraction; the "carry" status bit is set as follows on addition and subtraction:

high bit of A input	1	1	1	1	0	0	0	0
high bit of B input	1	1	0	0	1	1	0	0
high bit of output	1	0	1	0	1	0	1	0
carry on addition	1	1	0	1	0	1	0	0
carry on subtraction	0	1	1	1	0	0	0	1

The "carry" bit for other operations is meaningless.

These ALU status bits provide primitives for computing various status bits set by arithmetic and logical operations in a macromachine. In emulating macromachine operations which set status bits, the microcode would latch the ALU status on the appropriate microcode operation and then save the value of ALUST in a register or in main memory. If later the macromachines's status bits must then be tested, the saved value of ALUST can be decoded.

2.4 Program Control

Program execution is controlled by the Microcode Program Counter (UPC) and the Microinstruction Register (UIR). The UIR holds the instruction being executed, while the UPC holds the address of the instruction being fetched into the UIR for execution during the next cycle. In detail, suppose the UPC has value X at the start of microcycle M. During cycle M, the microcode instruction at address X is fetched into the UIR. Then during cycle M + 1, this instruction is decoded and executed.

The hardware normally increments the UPC at the start of each cycle, so that sequential instructions are fetched and executed. To effect a program branch, however, the microcode can load the UPC as a destination. (When the microcode specifies loading the UPC, actually both the UPC and another register, RAMADDR, are loaded. Section 2.10 explains RAMADDR.) Because microinstruction fetching is pipelined as described above, the transfer of control takes two cycles: on the second cycle following the UPC load, the instruction at the new address will be executed. To clarify, suppose that in cycle M the machine executes a microinstruction at address X which loads Y into the UPC. The following table then gives the contents of the UPC and UIR at the beginning of cycles M, M+1, and M+2:

Cycle	UPC	UIR
M	X+1	instruction at X (Y -> UPC)
M+1	Y	instruction at X+1
M+2	Y+1	instruction at Y

The intervening instruction at X+1, executed before control transfers to address Y, may be any otherwise legal microinstruction, including another transfer of control. In the above example, if the instruction at X+1 transfers control to address Z, the machine behaves as follows:

Cycle	UPC	UIR
M	X+1	instruction at X (Y -> UPC)
M+1	Y	instruction at X+1 (Z -> UPC)
M+2	Z	instruction at Y
M+3	Z+1	instruction at Z

If further Z = X+2, then the code has spliced an execution of the statement at Y into an otherwise sequential execution sequence.

Because the UPC may be loaded as a destination, the full data computation power of the machine is available for computing transfers of control.

Execution must never "fall through" from the lower 8K addresses to the upper 8K; Section 2.10 explains why.

2.5 Conditional Execution

Every microinstruction has a field for conditional execution. The following conditions are available:

always true}
ALU status: result zero
ALU status: result odd (low bit on)
ALU status: result negative (high bit on)
interrupt pending
mode flag 0 on
MAR condition

The ALU status conditions refer to the appropriate bits in the ALU Status Register. The "interrupt pending" condition is true if a microinterrupt request (Section 5) other than the programmable request is pending, and may change at any time. Mode flag 0, controlled by the application software, is a bit in the MISC register. The MAR condition is determined by the application-dependent MAR daughterboard.

The microinstruction may specify execution either when the selected condition is true or when it is false. By selecting the "always true" condition the instruction may specify either unconditional execution or unconditional non-execution.

When non-execution is specified, ALU performs the indicated operation on the indicated inputs, but the result goes nowhere: the hardware prevents writing to any destination or ALU scratch register, and also prevents latching the ALU status in ALUST.

Conditional execution is useful not only for program jumps but also for in-line execution.

2.6 Constants

The microinstruction may specify an explicit constant value to drive the source bus. Because the microinstruction field which specifies the constant is kept to a reasonable size, not all 20-bit constants may be specified. The instruction may specify any one of three 8-bit fields and can then choose only all 0's or all 1's for the other bits. The three fields are:

bits 0-7 (right), bits 8-15 (middle), and bits 12-19 (left). In addition to these six possibilities, the instruction may specify an "extended constant". In this case, bits 0-13 may be set as desired and the other bits are 0. Since the microcode address space only has 14 bits (16K words), loading an extended constant into the UPC can effect a program branch to any microcode address. When an extended constant is used, the ALU always does a PASS operation.

The assembler automatically encodes permitted constants into one of these seven forms, flagging illegal constants.

Any 20-bit constant C can be built in at most three cycles. Let

```
C1 = C AND     377
C2 = C AND  177400
C3 = C AND 3600000
```

and let R be an ALU scratch pad register. Then the instruction sequence

```
load C1 into R
load (C2 OR R) into R
load (C3 OR R) into ...
```

provides constant C.

2.7 Certain Coding Awkwardnesses

It is not possible to copy one scratch pad register to another, or to perform an ALU operation involving two scratch pad registers in a single microinstruction. Such operations take two cycles, requiring a register for temporary storage which is both a source and a destination. TEMP is often used for this purpose. Let R and S be registers. To copy the contents of R into S:

```
load R into TEMP
load TEMP into S
```

To add the contents of R and S:

```
load R into TEMP
load (S PLUS TEMP) into...
```

It is also not possible to combine two sources in a single microcycle. Such a combination takes two cycles, requiring a scratch pad register for temporary storage. For example, to add MIR to TEMP, a scratch pad register R is used as follows:

```
load MIR into R
load (TEMP PLUS R) into...
```

It is possible to subtract a source from a scratch pad register in one cycle, but not vice versa. To subtract a register R from a source S various techniques are possible, depending on where the result must go, whether R and S may be overwritten, and what constants are available in registers. For example, if the constant 0 is in some scratch pad register Z, and the result is to go to some register D (possibly S itself) which is both a source and a destination, the following two-cycle sequence works:

```
load (R MINUS S) into D
load (Z MINUS D) into D
```

2.8 MIR Daughterboard and Dispatch Memory

The Macroinstruction Register (MIR) is designed to hold macroinstructions. An application-dependent MIR daughterboard (MIRDB) helps to interpret the macroinstruction held in the MIR (see Figure 2.2).

The MIRDB can free the software from some cumbersome, frequently required calculations. As inputs to its computations, the MIRDB has: the 20 MIR bits, the four Mode Flags in MISC, and the microinstruction's eight-bit constant field.

The MIRDB has four functions. First, it provides a source, MIRFIELD, which is usually a modified form of the MIR. In emulating the SUE or PDP-11, for example, a good use for MIRFIELD might be to provide displacements on branch instructions. These machines compute the displacements by sign extending the low order byte and then shifting left one bit. In emulating these machines, performing such a calculation in software on every branch instruction would be tedious; some simple logic in the

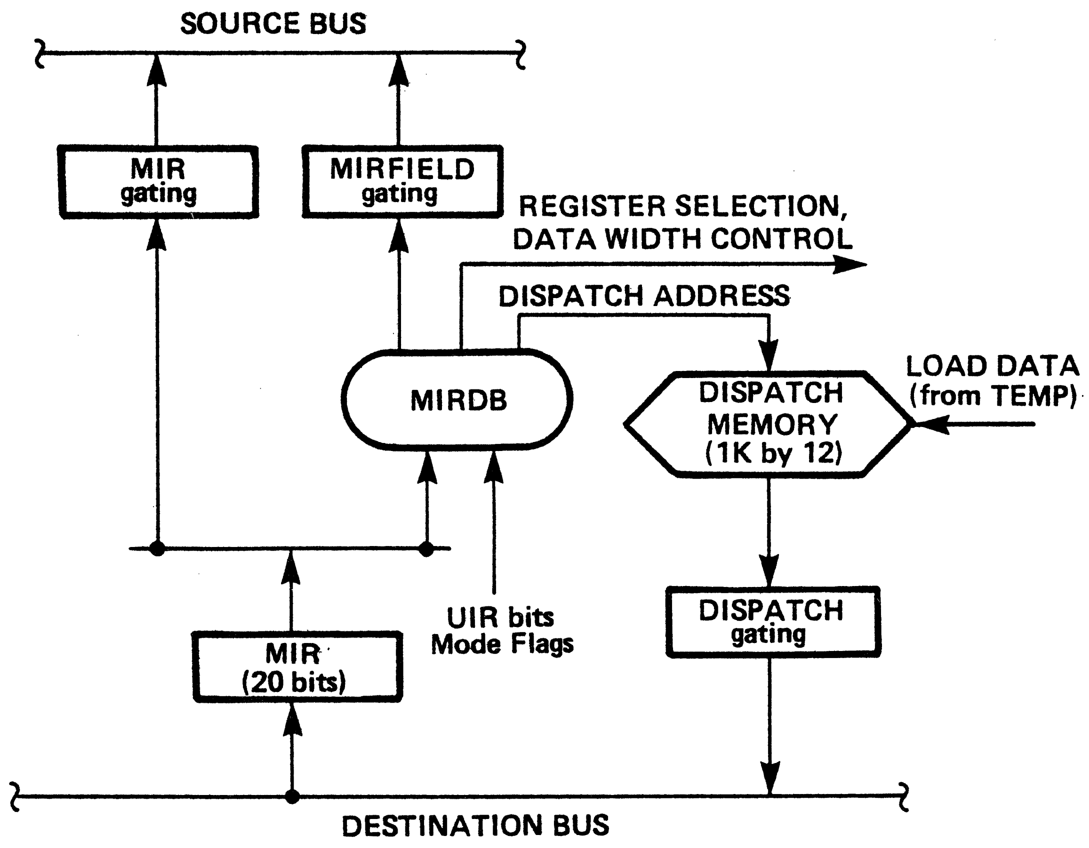


Figure 2.2: The MIRDB and Dispatch Memory
Picture 6

MIRDB, however, could provide the displacement in MIRFIELD.

Second, the MIRDB, under microcode control, can select ALU scratch pad registers. Values of 30 and 31 in the microinstruction's Register field indicate register selection by the MIRDB; the daughterboard generates two four-bit fields, corresponding to register values 30 and 31. When one of these two registers is selected, the register address supplied by the MIRDB is inclusive ORed with the contents of BASE to determine a scratch pad register address. This feature is useful for emulating macromachines whose instructions have fields specifying which of several macroregisters to use (such as index registers or accumulators). The macroregisters would be mirrored in a block of ALU scratch pad registers, and the macroinstruction's macroregister selection field would be used by the MIRDB to determine the corresponding ALU scratch pad register address. The application specific MIRDB helps solve the problem that these register fields are in different bit positions in different processors' instruction sets. The register selection function is performed frequently, and should be relatively efficient. Because of timing constraints, the MIR must not be used to select a register in the cycle immediately following an MIR load.

Third, the MIRDB can control the machine's data width mode (16 or 20 bits). This control is enabled when the DWIDTHCTRL bit in the MISC register is on. This feature might be useful for simultaneously emulating the instruction sets of a 16-bit machine and a 20-bit machine, or for providing an extension to the instruction set and memory of a 16-bit machine.

Fourth, the MIRDB, together with Dispatch memory, permits easy dispatching to the proper microcode routine to emulate different macroinstructions. When a microinstruction selects Dispatch as a source, the daughterboard calculates a 10-bit address. Dispatch memory is then interrogated at this address; the contents read are passed directly to the destination bus, and may be loaded into the UPC to effect a program branch.

The address computed depends on the contents of the MIR, the microinstruction's Constant field, and the Mode Flags in MISC. The MIRDB can let the microcode dispatch to an emulation routine which depends on the macroinstruction (MIR contents). Further, the microinstruction's Constant field lets the microcode specify different kinds of dispatches. For example, a macroinstruction's first dispatch might take all memory reference instructions to a common routine to calculate the effective address; memory reference instructions may then dispatch again, this time to separate routines to perform the indicated operation. The Mode Flags in MISC could reflect macromachine state bits which affect macroinstruction interpretation; the microcode could then dispatch to different macroinstruction emulation routines as appropriate. In the H316 emulation, for example, Mode Flag 0 gives the addressing mode ("normal" or "extended"), and the instructions JST, STA, IMA, and IRS have different emulation routines depending on the flag.

Dispatch memory is 1024 words by 12 bits. Its contents drive the destination bus directly because there is not enough time to send them through the ALU. The 12-bit value fetched from Dispatch memory is right justified in the destination bus. The MIRDB can specify the next two bits, allowing the memory values, when interpreted as microcode addresses, to reach all locations in the microcode address space.

Dispatch memory is itself loadable, permitting easy relocation of instruction emulation routines. Dispatch loading is enabled only when the LOADDISP bit in MISC is on. As long as the LOADDISP bit is on, any microinstruction which references an "odd" ALU scratch pad register (see next paragraph) risks loading a Dispatch location (even if it specifies conditional execution with a condition not satisfied); otherwise, the LOADDISP bit does not affect machine behavior. The low 12 bits of TEMP specify the data contents to be loaded. The MIR, the Mode Flags, and the microinstruction's Constants field specify the address to be loaded, in the same way that they specify the address to be read when Dispatch is selected as a source. To load a location, execute three microinstructions in a row. The first and third must reference an "even" register and the second must reference an "odd" register (see next paragraph). All three instructions must have the appropriate Constants field. Otherwise, these three instructions may be freely specified. (They may even specify conditional execution with a condition not satisfied.)

Every microinstruction specifies an ALU scratch pad register address in its Register field. We say that the instruction references an "even" or "odd" register according to whether the register address specified, _before any mapping by BASE_, is even or odd. Usually, bit 11 in the microinstruction (the low order bit of the Register Number subfield) determines this evenness or oddness: 0 for even, 1 for odd. However, if the Register field has value 30 or 31, specifying register selection by the MIR daughterboard, then the crucial bit is the low order bit of the four-bit field generated by the MIR daughterboard: again, 0 for even and 1 for odd.

To simplify loading, the MIR daughterboard should permit one value of the microinstruction's Constant field to specify a "transparent" mapping, in which the MIR's low 10 bits give the Dispatch address. By convention, a Constant field value of 0 will specify this transparent mapping. The current system software assumes this convention in the DDT code to examine and change Dispatch memory.

2.9 Shifting

The MAR and MBR form a double precision shift register, with the MAR on the left (high order). When the MAR is referenced as a source, the low two bits of the microinstruction's Constant field specify a shift operation:

00	no shift
01	shift right one bit
10	shift left one bit
11	undefined

Independent of the operation specified, the value of the MAR before shifting is loaded onto the source bus. The MAR and MBR shift only one bit per microinstruction, and 0 is shifted into the vacated bit; the burden of more complicated shift operations is on the microcode.

As shown in Figure 2.3, shifting respects the data width mode. In 20-bit mode, the MAR and MBR act as two 20-bit registers; in 16-bit mode, the MAR and MBR act as two 16-bit registers. Actually, the only difference is in driving MAR00 (bit 0 of the MAR) on left shifts, and in driving MAR15 and MBR15 on right shifts. In 16-bit mode, the top 4 bits of the MAR and MBR themselves shift, though they do not drive any other bits.

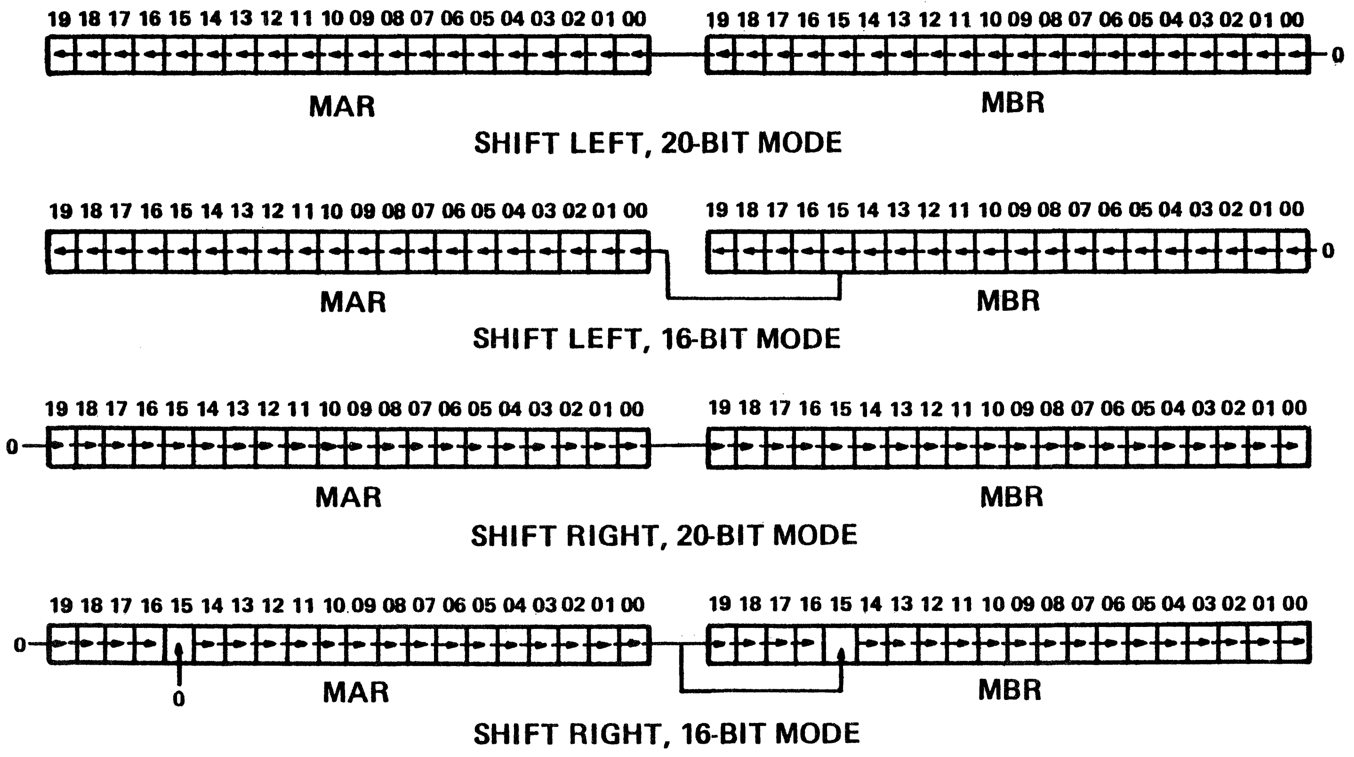


Figure 2.3: Shifting
Picture 7

2.10 Loading and Reading Microcode

The upper 8K of the 16K microcode address space may be loaded by the microcode, provided that the physical memory itself is loadable (RAM rather than PROM). To load the upper 8K, the program must be executing in the lower 8K. The low 16 bits of TEMP contains the contents to be loaded, and the 14-bit register RAMADDR (located on the microcode daughterboard) must contain the address to be loaded. Writing a 1 into the M2.LOADUH bit of MISC2 loads the high 16 bits of the microcode word; writing a 1 into the M2.LOADUL bit loads the low 16 bits. (Writing both of these bits at once loads both halves of the microcode word, each with the same contents.) If the bit is written in cycle m, then in cycle m+1 the memory's contents are indeterminate; by cycle m+2 the new contents are present. Further, cycle m+1 must not change the value of TEMP or RAMADDR.

Between writing the load address to RAMADDR and loading the microcode, the program must not load the UPC. (This restriction extends through the cycle after writing M2.LOADUH or M2.LOADUL.)

RAMADDR must not be loaded while executing in the upper 8K of the microcode address space. Further, the program must never "fall through" to the upper 8K of memory; any entry to the upper 8K must be by an explicit load of the UPC.

The hardware structure that imposes these restrictions is as follows. There are two 14-bit registers, UPC and RAMADDR. Specifying UPC as a destination loads both; specifying RAMADDR as a destination loads only RAMADDR. The high bit of UPC determines whether to fetch an instruction from the lower or upper 8K. If the lower half is used, the low 13 bits of the UPC give the rest of the address; if the upper half is used, the low 13 bits of RAMADDR give the rest of the address. Execution in the upper 8K increments both UPC and RAMADDR; execution in the lower 8K increments only UPC. When executing in the lower half, RAMADDR, not being incremented, is free to be used for specifying a loading address. The M2.LOADUH and M2.LOADUL bits enable loading, which takes two cycles. Since loading the UPC would overwrite any load address in RAMADDR, loading the UPC must be avoided during microcode loading. Execution must never "fall through" to the upper 8K since RAMADDR would not be in step with the UPC.

The microcode parity bit must be explicitly loaded as desired (to give the word odd parity), as part of the low 16 bits. The hardware does not generate parity.

Reading microcode memory also uses RAMADDR. Only the upper 8K may be read, and the reading must be executed in the lower 8K. If RAMADDR contains address x at the start of cycles m and m+1, then in cycle m+1 the sources URAMH and URAML yield the contents of the high and low 16 bits, respectively, of the location at address x. Thus, the standard reading sequence is: load RAMADDR, wait one cycle, and read URAMH or URAML (or one after the other).

2.11 Miscellaneous Features

To ease byte operations, two byte-swapped forms of the MBR (called S16MBR and S20MBR) are available as sources. When S16MBR is specified, the low 8 bits are exchanged with the middle 8 bits (the order within each byte staying the same); the top 4 bits stay in place. When S20MBR is specified, the low 10 bits are exchanged with the high 10 bits. The MBR was chosen because it is a read/write register which serves as a data buffer for main memory and I/O operations, and desired byte operations will likely deal with data transferred to or from main memory or I/O devices.

A "serial number" source yields a fixed 16-bit constant. The low order 12 bits, determined by cuts in PC board etchings, hold the machine's unique serial number. The high order 4 bits, determined by solder wire jumpers on the PC boards, represent the assembly revision level of the machine.

2.12 Handling Exceptional Conditions

On detecting various exceptional conditions, the hardware traps microcode execution to address 0. The PROM memory starting at address 0 should have an appropriate handler. The conditions detected include new power, microcode parity error, uncorrectable main memory error, and button pushed.

When power is restored, the hardware forces execution to address 0 and holds it there for roughly a half second to let the hardware settle.

Microinstructions should have odd parity. When an instruction with even parity is fetched, the hardware detects an error and signals for a trap to address 0. Subject to the caveat below on trap multiplexing, the trap is timed so that execution branches to 0 after the instruction following the instruction with the bad parity.

If the main memory Error Detection and Correction (EDAC) option is present, the hardware senses certain uncorrectable errors in main memory (Section 4). If EDAC trapping is enabled, the EDAC logic will then signal for a trap to address 0. Subject to the caveat below on trap multiplexing, the trap is timed so that execution branches to 0 after the second cycle in which the fetched value is available. (Section 3 discusses memory access timing, including when fetched values becomes available.)

A manual push-button is located on the processor card. When the button is pushed, the hardware signals for a trap to address 0. This is used to unilaterally enter the microcode DDT.

The conditions of microcode parity error, uncorrectable main memory error, and button being pushed are actually multiplexed to one trap generator. The trap is generated by the rising edge of the inclusive OR of the microcode parity error condition, the main memory error condition, and the button being pushed. The microcode parity error and main memory error conditions are asserted for only one microcycle on each error, but the button condition is asserted as long as the button is pushed; as long as the button remains pushed, microcode parity errors and main memory errors cannot cause traps. The trap's mechanism is to force the microprogram counter to 0 for several cycles. (The power-up trap uses the same mechanism, but it forces the microprogram counter to 0 for much longer.)

Various status bits let the handler at address 0 determine what conditions have occurred. A bit in register MISC2 reflects whether the button is being pushed; other bits in MISC2 are latched upon new power, a microcode parity error, and an uncorrectable main memory error.

3 MAIN MEMORY

Main memory, like the ALU scratch pad register file, holds read/write data. Compared with the ALU register file, main memory is much larger but somewhat slower. Main memory is intended to contain the macromachine's address space, holding the macromachine's program and data. Main memory may also store information for the microcode's private use.

Main memory is implemented with dynamic RAMs. These are much larger than the static RAMs used elsewhere in the MBB, and have a much longer access time. Dynamic RAMs have special constraints which must be accommodated by the microcode, such as refreshing and power up "priming". These are discussed later.

The Memory Buffer Register (MBR) buffers data, and the Memory Address Register (MAR) specifies the memory address (see Figure 3.1). Whenever the MAR is loaded, a four-bit transfer field is latched into an auxiliary "memory operation" register (called MAROP). The value latched specifies whether or not to initiate a memory or I/O transfer and, if so, whether to read or write and whether to use "physical" or "macro" access (discussed below).

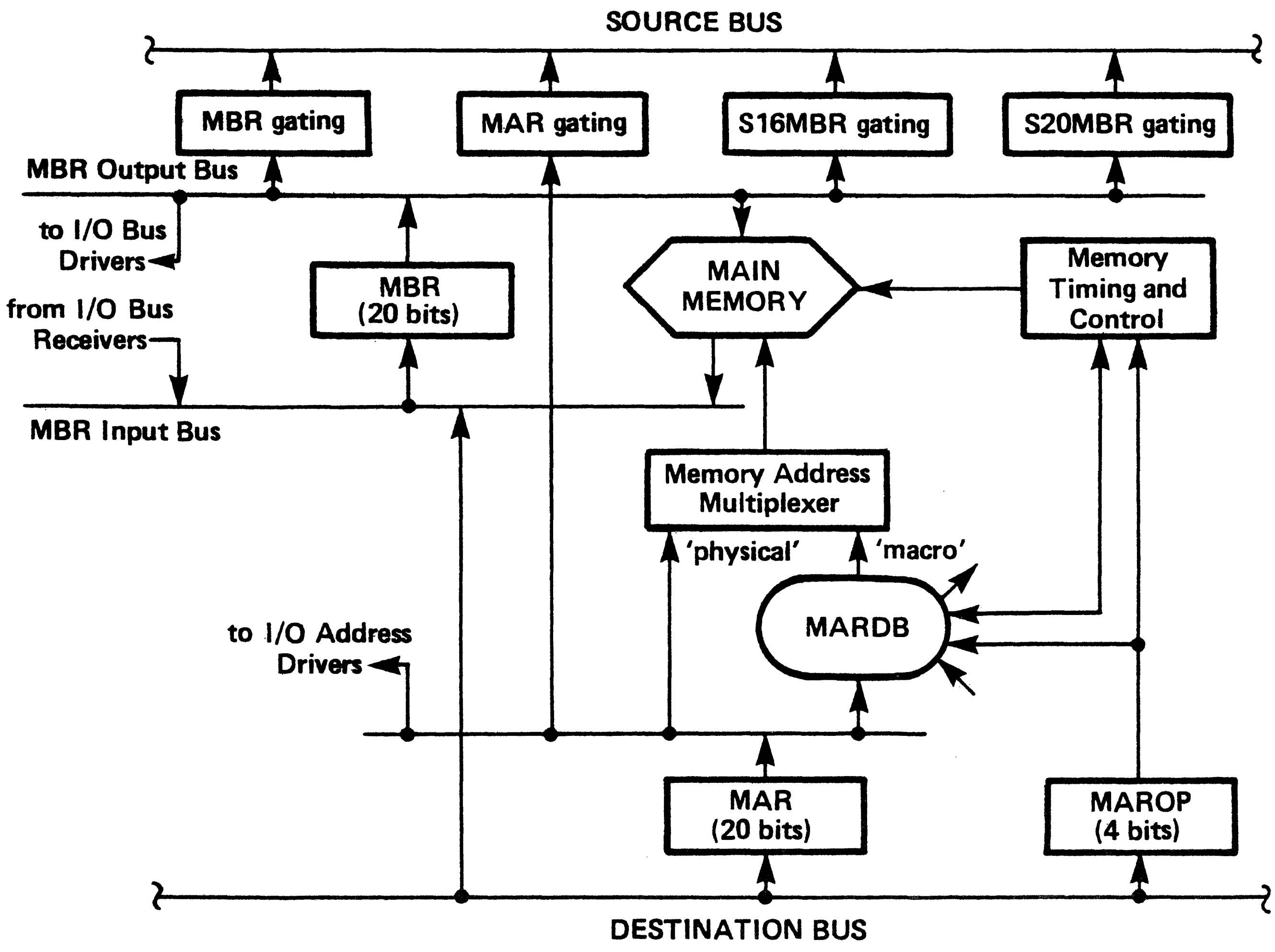


Figure 3.1 Main Memory Address and Data Paths
Picture 8

3.1 Read and Write Paradigms

By convention, cycles are numbered starting with the cycle after the MAR load. The read paradigm for a "physical" memory access is as follows:

```
        load address in MAR (Transfer code says "read")
[1]     don't change MAR, MBR unchanged
[2]     don't change MAR, MBR unchanged
[3]     memory contents in MBR, MAR may be changed
```

Cycles [1] and [2] may be used freely, provided the MAR is unchanged. In cycles [1] and [2], the MBR retains its old contents. In cycle [3], the MBR contains the fetched memory contents, and the MAR may be loaded to initiate another memory operation.

The write paradigm for a "physical" memory access is as follows:

```
       load contents into MBR
       ...
       load address into MAR (transfer code says "write")
[1]    don't change MAR or MBR
[2]    don't change MAR or MBR
[3]    main memory has been changed
```

Again, cycles [1] and [2] may be used freely except to write the MAR or MBR, and cycle [3] may start a new transfer. Any number of cycles may intervene between the MBR load and the MAR load.

3.2 Memory Timing

Main memory transfers take more than a single microcycle. When nested as closely as possible, transfers in general take N cycles each; with the current hardware, N is 3. To change the paradigm to fit a memory where N is greater than 3,* simply expand cycles [1] and [2] into N-1 cycles, with the same constraints on each one. The fetched contents on reads are then available in cycle [N]. In the general case, an uncorrectable main memory error detected by the optional Error Detection and Correction (EDAC) logic will trap to execute at microcode address 0 during cycle [N+2]. Section 2.12 discusses traps to address 0, and Section 4 discusses EDAC.

3.3 Memory Access Modes

On each read and write, the transfer code latched into MAROP specifies an access mode. Access may be "physical" or "macro". For a "physical" access, the address is passed directly to the memory from the MAR. For a "macro" access, the address is obtained from the application-dependent MAR daughterboard (MARDB), as shown in Figure 3.1. The daughterboard may perform memory address translation, either simple schemes (like byte-

----
*For example, the overhead of the UNIX memory management MARDB adds one microcyle to every "macro" main memory access; here N is 4.

word) or full scale memory mapping. Further, the MARDB may treat some values in the MAR as specifying illegal addresses, and prevent any memory operation from occurring. In addition, the MARDB can control a condition MARCOND, which the microcode can test with conditional execution (Section 2.5).

The "physical" access is intended for handling the microcode's private data area, while the "macro" access is intended for emulating a macromachine's memory operations. The MAR daughterboard can emulate the macromachine's memory mapping, and can ensure that macromachine operations never reference the microcode's reserved area of main memory. As input for its decisions, the MARDB has the 20 MAR bits, the four MAROP bits, and the four Mode Flags in MISC. (In the H316 emulation, one Mode Flag bit indicates "normal" or "extended" addressing mode, and a second bit indicates "upper" or "lower" bank if in "normal" mode; the MAR daughterboard uses both flags.)

In some machines (such as the SUE), the same instruction codes reference memory and I/O devices; some addresses simply point to I/O devices rather than to actual memory. To emulate such a machine, memory reference macroinstructions must be handled differently depending on the address. The MAR daughterboard could help by setting the MARCOND condition according to whether the address loaded in the MAR indicates memory or I/O. (Presumably the MARDB would also prevent main memory access if the address indicated I/O.)

By providing some basic application-dependent hardware capabilities, the MAR daughterboard can free the microcode from common, tedious calculations.

3.4 Special Considerations

Main memory uses dynamic RAMs, which must be periodically "refreshed" to maintain their contents. For the MBB, this refreshing consists of reading the memory. The scope of reading needed depends on the chip size. If a chip contains 2N bits of addressing, the low N bits are called the "row" bits and the high N bits are called the "column" bits. For every combination of row bits, that combination must occur in a main memory read at least every 2 milliseconds. Currently 16K (14-bit address) RAMs are used, so there are 2**7 (128) reads to perform. (The total number of chips used does not matter, since all chips are refreshed in parallel.)

After power-up, the dynamic RAMs must be accessed (either read or written) a minimum of eight times before they will function properly. This "priming" of main memory is done automatically by the system PROM microcode.

4 MAIN MEMORY ERROR DETECTION AND CORRECTION (EDAC)

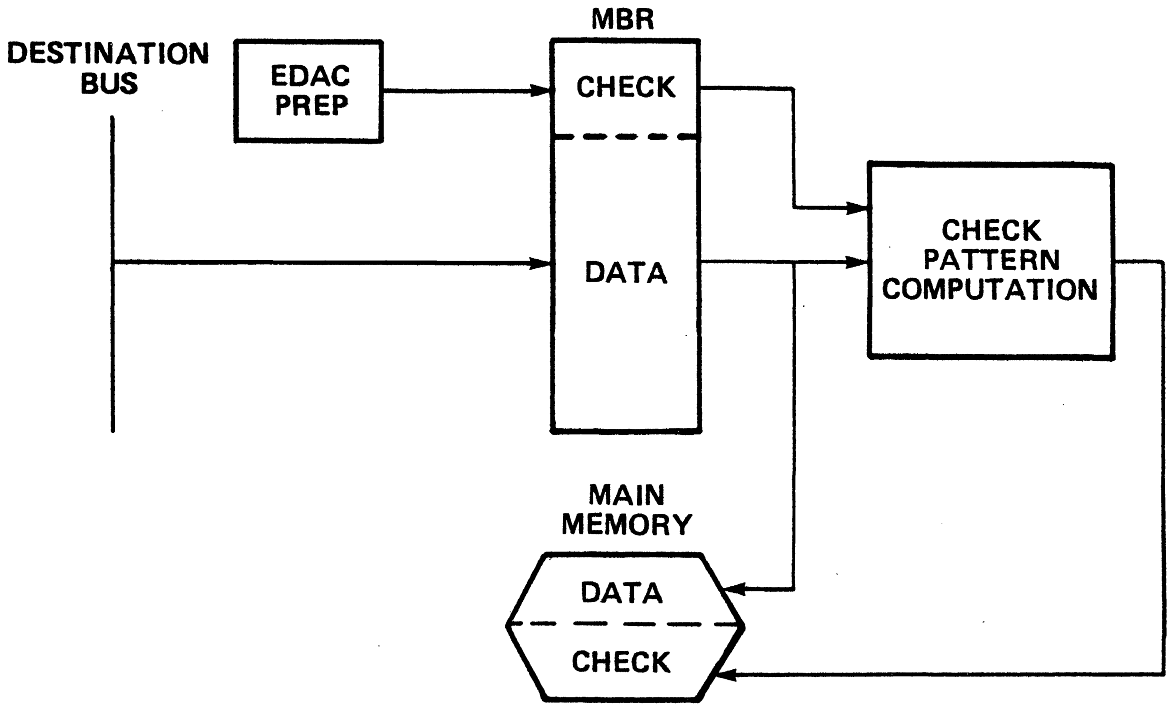
The MBB uses dynamic semiconductor memory devices for its main memory. Because these devices may give occasional bit errors, the MBB provides special error detection and correction logic (EDAC) as an option. If the EDAC option is present, main memory and the MBR are 26 instead of 20 bits wide. The extra 6 bits, called "check" or "syndrome" bits, provide redundant information used to detect and correct bit errors. All single bit errors can be corrected; all double bit errors can be detected; errors in more than two bits are generally either not detected or improperly corrected. We first describe the algorithm used to detect and correct errors, and prove the algorithm's correctness; we then explain how EDAC is integrated into the MBB. Figure 4.1 summarizes EDAC data paths and operation.

4.1 The EDAC Data Paths

Both main memory and the MBR are 26 bits wide: 20 data bits, plus 6 check bits. When a word is written from the MBR to memory or read from memory to the MBR, the data bits are passed without change.

When a word is written from the MBR into memory, a 6-bit "write check pattern" is computed and stored in the memory

WRITE ACCESS



READ ACCESS

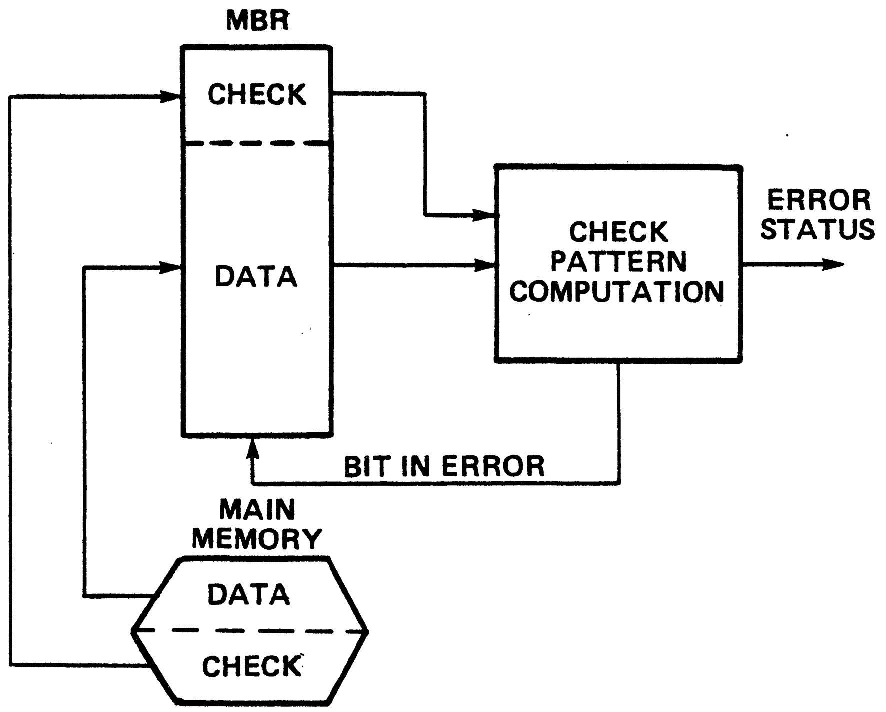


Figure 4.1: EDAC Data Paths
Picture 9

location's check bits. When a word is read from memory into the MBR, the memory's check bits are copied into the MBR; a 6-bit "read check pattern" is then computed to detect and correct bit errors. Both the write check pattern and the read check pattern are functions (described below) of the 26 bits in the MBR; essentially the same logic is used to compute the two patterns.

4.2 The EDAC Algorithm

Two "boundary conditions" are assumed for memory use: no memory word is read without having first been written, and before a write the MBR's check bits all have value 1. In this section these conditions will enter our proof at peripheral points; in the next section their "physical meaning" will be discussed.

The check bits' values in the MBR and memory, as well as the read and write check patterns, can be represented 6-bit strings such as "011111". From left to right, these bits will be called the "parity" bit P and the "syndrome" bits S4, S3, S2, S1, and S0. The data bits in the MBR and memory are labeled D19 through D00 (D00 is the low order bit). Lower case p and s denote bit values while upper case P and S denote the bits themselves.

The check pattern computation functions can be determined from Table 4.1. Each column is associated with a certain check bit, and each row is associated with a certain check or data bit in the MBR (or is "unused"). For any check bit R, define c(R) (the set of bits "covered" by R) as the set of check and data bits B such that the entry in R's column and B's row is an "*". Define c'(R) to be just the data bits in c(R). Thus, for example, c(P) is the set of all 26 bits, since P's column contains all *'s; and c'(P) is the set of all data bits.

P	S4	S3	S2	S1	S0	
*	-	-	-	-	-	P
*	-	-	-	-	*	S0
*	-	-	-	*	-	S1
*	-	-	-	*	*	unused
*	-	-	*	-	-	S2
*	-	-	*	-	*	unused
*	-	-	*	*	-	unused
*	-	-	*	*	*	unused
*	-	*	-	-	-	S3
*	-	*	-	-	*	unused
*	-	*	-	*	-	unused
*	-	*	-	*	*	D19
*	-	*	*	-	-	D18
*	-	*	*	-	*	D17
*	-	*	*	*	-	D16
*	-	*	*	*	*	D15
*	*	-	-	-	-	S4
*	*	-	-	-	*	D14
*	*	-	-	*	-	D13
*	*	-	-	*	*	D12
*	*	-	*	-	-	D11
*	*	-	*	-	*	D10
*	*	-	*	*	-	D09
*	*	-	*	*	*	D08
*	*	*	-	-	-	D07
*	*	*	-	-	*	D06
*	*	*	-	*	-	D05
*	*	*	-	*	*	D04
*	*	*	*	-	-	D03
*	*	*	*	-	*	D02
*	*	*	*	*	-	D01
*	*	*	*	*	*	D00

Table 4.1: EDAC Check Bit Computation

The check bits are computed in the same way for the read and write check patterns. The value for a check bit Sn (n=0,1,2,3,4) in the check pattern is the complement of the EXOR of the values in the MBR of all bits in c(Sn).

Similarly, the value for P in a read check pattern is the complement of the EXOR of the values in the MBR of all bits in c(P). The value for P in the write check pattern, however, is the EXOR of the value of P in the MBR, the values of the bits in c'(P) in the MBR, and the values of the check bits in the _write check pattern itself_.

Since c(P) includes all 26 bits and P has value 1 in the MBR before a write, we can restate the computation of P in the check patterns as follows. In the write check pattern, P gives the word written (data plus check bits) an odd parity. In the read check pattern, P reflects the parity of the word read: 1 for even parity, 0 for odd parity.

We will study first the check patterns generated according to these rules. Later we will analyze the effect of hardware errors.

We will show that, if a word is written into memory and then read back without error, the read check pattern is always 011111. Let p, s4, s3, s2, s1, and s0 be the values of the bits in the read check pattern; let p', s4', s3', s2', s1' and s0' be the values of the bits in the write check pattern (which are stored in memory during the write); and let p", s4", s3", s2", s1", and s0" be the values of the check bits in the MBR before the write. (p", s4", s3", s2", s1", and s0" are normally 1's, but we defer using this assumption in order to make our result more general for later use. Since all data bits have the same value throughout the transfer, we need not define separate symbols for their values before the write, after the write, and after the read.) If a and b are bit values, let a * b denote a EXOR b and let a denote the complement of a; if A is a set of bits, let *A denote the EXOR of the values of all bits in A.

We can express succinctly with equations the rules given above in prose for calculating the read and write check patterns. The read check pattern is computed from values in memory as follows:

(1) $p = ((*c'(P)) * p' * s0' * S1' *s2' * s3' * s4')$

(2) $sn = (*c'(Sn) * sn')$.

(We use n to stand for 0, 1, 2, 3 or 4. Thus, the second equation is really five equations, one for each check bit. Equation (2) holds because Sn covers itself but no other check bit; equation (1) holds because P covers every check bit.) The write check pattern is computed from the values in the MBR as follows:

(3) $p' = (*c'(P)) * p'' * s0' * s1' * s2' * s3' * s4'$

(4) $sn' = ((*c'(Sn)) * sn'')$.

We can substitute (3) and (4) into (1) and (2) to yield the read check pattern values as functions of the original values in the MBR. Substituting (3) into (1) yields

(5) $p = (p'')$

and substituting (4) into (2) yields

(6) $sn = sn''$.

Thus, the read check pattern depends only on the initial check bit values in the MBR, not on the data bits. We are assuming an initial pattern of 111111; (5) and (6) thus yield a read check pattern of 011111.

So far, we have shown that an error-free "round trip transfer" from MBR to memory and back to MBR yields the read check pattern 011111. We will now examine the read check patterns caused by various hardware errors.

Our model of a hardware error is that in memory, between the write and the read, one or more bits change value. P's behavior on errors is easy to analyze. P in the read check pattern reflects the parity of the word read. In the absence of errors, P in the read check pattern is 0, reflecting the memory word's correct odd parity. In general, P in the read check pattern reflects the number of bits in error in memory: P is 0 if the number of errors is even (including none), and 1 if the number of errors is odd.

Having completely analyzed P's behavior with errors, we now consider that of the check bits. If only one bit in memory is in error, just those check bits which cover the erroneous bit will appear as 0 in the read check pattern instead of the normal 1. For example, if D19 is in error, since just S3, S1, and S0 of the check bits cover D19, the check bits will appear as 10100 in the read check pattern. From this example, we see that the bit in error can be read from Table 4.2 by matching the parity and check bits in the read check pattern. (The EDAC.BANK field of the EDAC register must be examined to uniquely determine the failing memory chip.)

If exactly two bits are in error, then, since the two bits cannot be covered by exactly the same syndrome bits, the syndrome bits in the error pattern will not be 11111.

We can summarize the read check patterns for zero, one, and two errors:

Read Check Pattern	Bit in Error	Physical Location
100000	MBR00	M1, M27
100001	MBR01	M13, M39
100010	MBR02	M14, M40
100011	MBR03	M26, M52
100100	MBR04	M2, M28
100101	MBR05	M12, M38
100110	MBR06	M15, M41
100111	MBR07	M25, M51
101000	MBR08	M3, M29
101001	MBR09	M11, M37
101010	MBR10	M16, M42
101011	MBR11	M24, M50
101100	MBR12	M4, M30
101101	MBR13	M10, M36
101110	MBR14	M17, M43
101111	S4	M20, M46
110000	MBR15	M23, M49
110001	MBR16	M5, M31
110010	MBR17	M9, M35
110011	MBR18	M18, M44
110100	MBR19	M22, M48
110111	S3	M21, M47
111011	S2	M19, M45
111101	S1	M8, M34
111110	S0	M6, M32
111111	P	M7, M33

Table 4.2: Read check pattern following a single-bit error

<u>Number of Errors</u>	<u>Read Check Pattern</u>	
0	011111	
1	1xxxxx	(xxxxx gives bit in error)
2	0xxxxx	(xxxxx is not 11111)

Single errors can be detected by P being 1, and corrected using the check bits' value. Double errors can be detected by both P and at least one check bit being zero. The EDAC logic interprets any read check pattern as no error, a single error, or a double error. If more than two bits are in error, an odd number of errors will be interpreted as a single error, and an even number of errors will be interpreted as no error or a double error.

Since no memory location is read without first having been written, every read from memory is the second half of a "round trip transfer". Therefore, the EDAC logic can safely perform this error analysis on every read.

4.3 Integrating EDAC into the MBB

The discussion of the EDAC algorithm assumed that the MBR's check bits were all 1's before a write to memory. When the MBR is loaded from the destination bus, the check bits are loaded from the 6-bit EDAC.PREP field of the EDAC register (Section 7.4). For normal operation, this field should be set to 111111. For hardware debugging, it may be given another value. When the MBR is loaded by a memory or I/O fetch, the check bits generally are loaded with different values. (A memory fetch, of course, copies the check bits from memory. An I/O fetch loads the check bits randomly.) Therefore, before any main memory write, the last load of the MBR must be from the destination bus. In the following memory-to-memory transfer sequence, the statement "MBR->MBR" copying the MBR to itself through the ALU, is needed for this reason.

```
LOC1->MAR(R)        ; read main memory at address LOC1
...
...
MBR->MBR            ; fix MBR's check bits
LOC2->MAR(W)        ; write value to main memory
                      at address LOC2
...
...
```

Equations (5) and (6) of the last section show that in the error-free case, toggling any EDAC.PREP bit in the EDAC register simply toggles the corresponding bit in the read check pattern. For example, having 010111 instead of 111111 in the EDAC register produces a read check pattern in the absence of hardware errors of 110111 instead of 011111. This would be detected as an error by the EDAC hardware. (Errors will work much as before, changing bits in the read check pattern away from their values in the error-free case.)

The discussion of the algorithm also considered only those reads which follow writes. In fact, the EDAC logic would be confused on reading a location which had never been written, since the check bits in memory would be random. Therefore, as part of the initialization after power-up, all main memory locations should be written. (Such writing must be preceded by the eight memory cycles required to prime main memory.)

On detecting errors after reads, the EDAC logic affects MBB operation in three ways. First, various status conditions and error data are latched. On detecting any error, the "M2.MEMERR" bit in MISC2 is set. On detecting any double (uncorrectable) error, the "M2.UCERR" bit in MISC2 is set. On detecting any single (correctable) error, the read check pattern and the memory bank from which the erroneous data came are saved in the EDAC register. Thus, faulty bits in individual chips may be identified.

Second, if enabled, EDAC corrects single data errors. On detecting a single data error, EDAC steals two microcycles to fix the indicated bit in the MBR. EDAC does not eliminate the error in memory, so every subsequent fetch would require the same fixing. (The program should eventually eliminate the error by rewriting the corrected data to memory.) Setting the "EDAC.FIX" bit in the EDAC register enables the cycle stealing and MBR fixing; clearing the bit disables them.*

----
*Note that if fixing is not enabled, the read check pattern is _not_ latched in the EDAC register on an error.

Third, if enabled, EDAC traps to microcode address 0 on detecting a double error. Setting the "EDAC.TRAP" bit in the EDAC register enables this feature; clearing the bit disables the feature.

The current system software (see Chapter 8) leaves fixing enabled always, but leaves traps enabled only when running application code. The program listing gives details.

5 I/O and INTERRUPTS

5.1 I/O Design Issues

Before discussing the details of I/O access and operation, a brief description of some facets of MBB I/O philosophy is in order.

5.1.1 Hardware/Software Tradeoffs

A key goal of the MBB design is to minimize the hardware devoted to I/O interfaces. MBB I/O hardware performs only very basic tasks such as handshaking, electrical level conversion, serial-parallel conversion, and requesting a microinterrupt after transferring a unit of data. The microcode then has responsibility for such functions as maintaining the finite state machine for the line, transfer of data to or from memory (DMA), status reporting, word assembly/disassembly, padding, and checksumming. Such microcode is considerably cheaper and easier to supply and change than the corresponding hardware.

I/O emulation is not a trivial demand on the micromachine. In a communications application, such as the MBB IMP, half of the processor bandwidth is nominally budgeted to support the I/O system, leaving the other half for instruction emulation. This bandwidth tradeoff is, of course, dynamically allocated based on the instantaneous I/O processing requirements of the machine.

As an example, it is helpful to consider the actions involved in the receipt of a data character on a communications line ("modem") interface. When the interface shifts in 8 bits, a microinterrupt is generated. When this interrupt is serviced, the microcode reads the data byte from the interface and references that device's status block in register memory. This area contains:

- finite state machine state
- residual characters
- checksum
- DMA pointers
- error and status indicators

If the state information indicates the interface is idle, the character is checked to see if it is a DLE (the transparency escape character). If the interface is in data mode, the checksum calculation is updated via table lookup (using tables stored in reserved main memory). Then this character is combined with the previous one (saved in the status block) to form a word. The DMA pointers are updated and the word written to memory. Control of the micromachine then passes to the next task. This action continues for the remainder of the packet, with the microcode detecting the end of message sequence and checking the checksum. Status information is updated and the completion macrointerrupt is requested.

5.1.2 The Microinterrupt System

The I/O service request system of the MBB is a hybrid of classic polling mechanisms, classic vectored interrupt systems and the Pluribus PID. The interrupts do not in fact "interrupt" the processor; rather, the processor must periodically poll in order to service any pending interrupt. Hardware is provided, however, to synchronize interrupt requests, to perform priority ordering, and to generate an interrupt vector. It is the microprogrammer's responsibility to access this hardware often enough to meet latency requirements. If no interrupt request is pending, the interrupt vector takes the MBB to the start of emulation of the next macroinstruction.

This approach has several advantages over real interrupts. First, the hardware is simpler and overhead is lessened since it is not necessary to automatically save and restore the UPC and other micromachine context. Second, it eliminates the mechanism of enabling and disabling microinterrupts. This would require at least one extra bit in the microinstruction since it would be too expensive to execute many enable and disable microinstructions. Finally, this approach should considerably simplify the design of the microcode since interrupts are more readily controlled.

5.2 I/O Access and Data Transfer

As explained earlier, I/O devices are accessed much like main memory. The MAR specifies the address, microinstruction bits determine the nature of the transfer (I/O or memory; read or write), and the MBR serves as a data buffer.

As shown in Figure 5.1, the "I/O Bus" signals include an I/O address bus with qualifying bits, and a data bus with control signals. The address bus has 16 bits and is driven from the MAR. The address qualifying bits are the bits from the microinstruction mentioned above. The data bus is tri-state and bi-directional. On output, the MBR drives the bus with data for a device; on input a device drives the bus with data for the MBR. Paths are provided for 20 bits. For devices which use fewer than 20 bits, high order MBR bits are ignored on output and 0's are placed in the MBR on input. To ease character-oriented manipulation of received data, data loaded from devices into the MBR can be gated subsequently onto the processor's source bus in regular or byte-swapped form.

The control aspect of interfacing to I/O is intended to be flexible; we must allow for a large variety of devices to be connected and still provide an efficient coupling with the micromachine. One significant variation is the response time of various devices: An interface constructed of TTL can generally respond to a request within one cycle while typical LSI devices

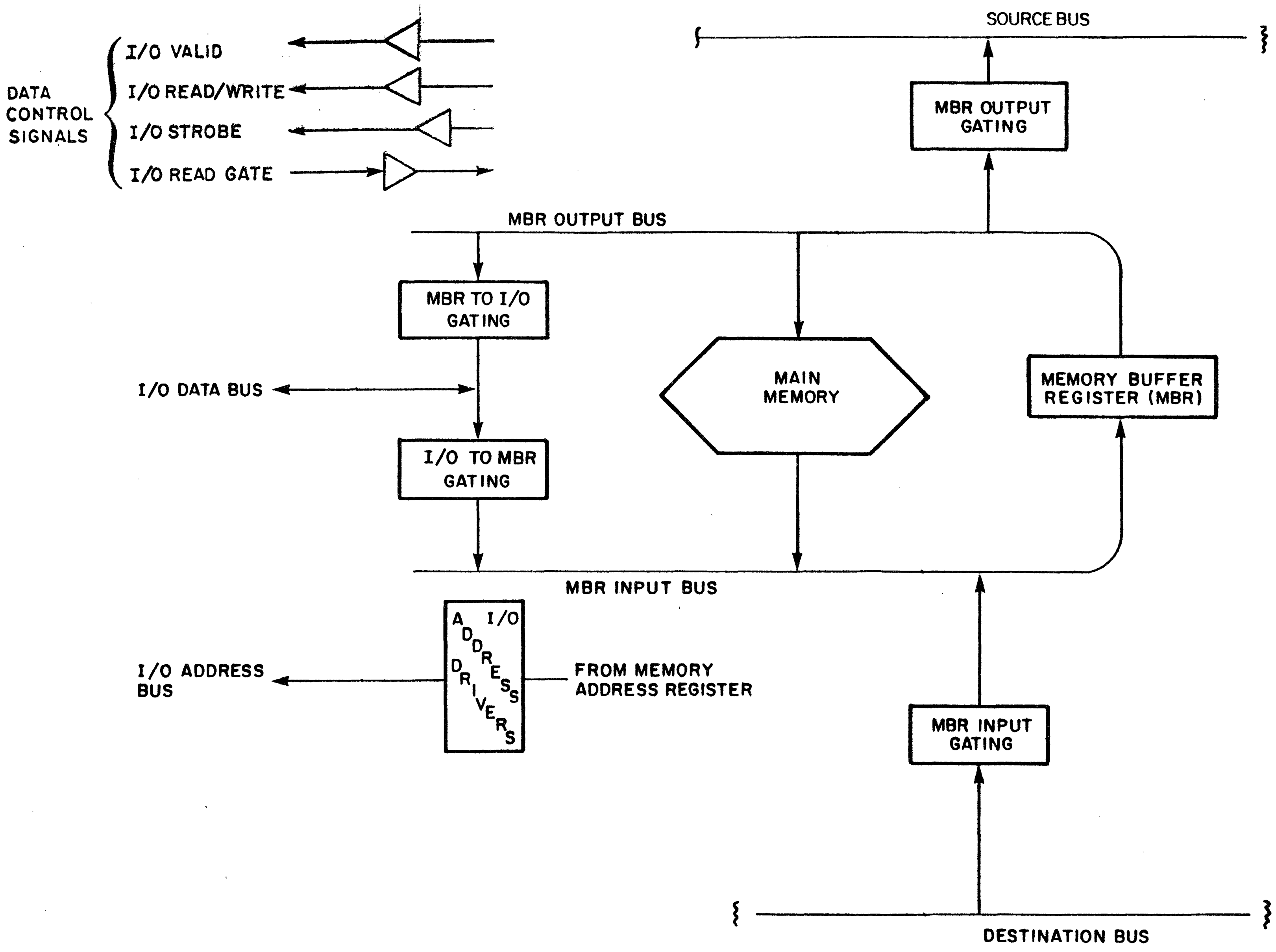


Figure 5.1 I/O Data Transfer
Picture 10

(e.g., a USART or floppy disk controller) have required input pulse widths and data access times which amount to several MBB cycles.

The resultant control mechanism has four signals. The first, driven by the MBB, says whether the access is read or write. This bit, called IOWRITE, controls the bidirectional I/O data bus drivers and is sourced by bit 1 of MAROP.

The second signal, called IOSTROBE and driven by the MBB, says "do it!". After loading the MAR with an I/O address and (if a write) the MBR with the data, IOSTROBE is asserted for N consecutive cycles, where N is dependent on the time constants of the actual device being addressed. IOSTROBE is asserted by setting the M2.IOS bit in MISC2. Note that this bit does not need to be explicitly cleared; IOSTROBE is on for the length of a cycle if, and only if, M2.IOS was set by the previous microinstruction.

The third signal is driven by each I/O board. It is used only by the hardware; the microcode cannot directly discover its state. It is asserted on a read to say that the data on the I/O Bus is valid. The MBB uses this to latch the data into the MBR. The I/O board generates this signal at the appropriate time, relative to IOSTROBE, knowing the time constants involved for the device addressed.

It is the microcode's responsibility to know the time constants for the devices it accesses, both the number of cycles IOSTROBE must be asserted and in which cycle the read data will be valid.

The fourth signal, IOVALID, resets all I/O devices, either at power-up or when the bit M2.IORESET is set in MISC2. (Note that M2.IORESET only affects I/O devices actually on I/O boards; the "standard" devices are reset only by power-up.) It is the responsibility of every I/O board designer to insure that each I/O board can be reset to a known state by removing IOVALID for a single clock cycle.

5.3 I/O Address Space

I/O devices* have 16 bit addresses. The high order three bits of an address denote the I/O board on which the device resides (see figure 5.2). I/O boards are numbered by their physical position in the box. The devices on the MBB's memory board (the "standard" devices) are considered to be on board zero; the I/O board cabled to the memory board is board one, etc. (There is thus a maximum of seven I/O boards in any MBB system). The other 13 bits of a device address have no a priori meaning;

----
*A "device" may be a communications interface, a control and status register, or some other unit which can either supply data to or accept data from the micromachine.

each I/O board has its own specific assignments.

5.4 Standard MBB I/O Devices

The MBB provides two communications paths, suitable for a terminal, a cassette loader, or a computer port. The standard I/O devices include these two interfaces (called individually the "console" and "loader" interfaces and collectively the "terminal" interfaces), a (read-only) switch register, and an Interrupt Enable Register. (The current system software controls these devices, as explained below, in Section 10.1, and in the program listing.)

The two terminal interfaces are each built around a USART (Signetics 2651) which handles 8-bit transfers to and from the processor. (For more detail than that given here, see the listing and the USART's data sheet.) The current system software configures these USARTs to function as full duplex, asynchronous interfaces. The MBB's processor may access four different registers in the USART; the "device's internal register" field in the I/O address specifies which of the USART's registers to use. (The MBB's addressing mirrors the USART's internal numbering.) See Figure 5.2.

Writing to Register 0 gives the USART data to transmit; reading from Register 0 reads the USART's received data. Writing to Register 1 sets line protocol synchronization characters, necessary only if the USART is configured as a synchronous interface; reading from Register 1 reads status such as ready and overrun conditions for transmission and reception. Register 2 is used to read and write configuration information such as synchronous/asynchronous operation, baud rate, and treatment of data parity. Register 3 is used to read and write command information such as enabling of transmission and reception, and data looping.

5.4.1 Local Interrupts

The USART buffers one byte in each direction in addition to the byte being transmitted or received. It sends interrupt signals to the processor when it is ready to provide more received data or to accept more data to transmit. The microcode can disable the transmit and receive interrupts by disabling the USART's transmitter and receiver respectively. (The current system software disables the transmitter when it has nothing to send; the software can leave the receiver enabled without interruption, since it is always ready for a receive interrupt.)

Since the receive and transmit interrupts indicate respectively received data ready and room for data to transmit, reading received data and supplying data to transmit will turn off these interrupts. When a byte is read or written there is a

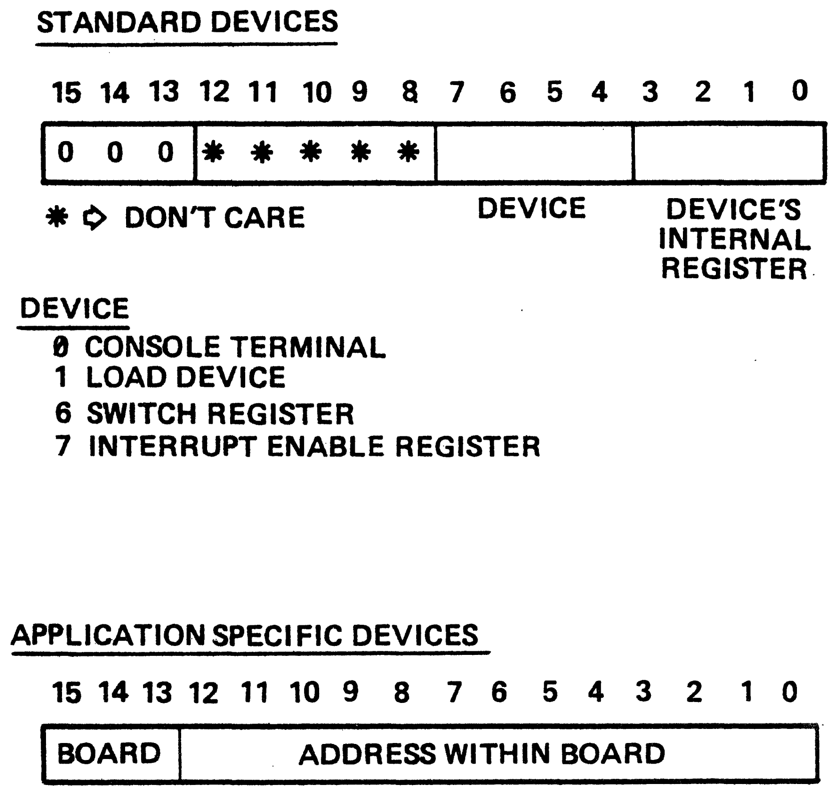


Figure 5.2 I/O Device Addressing
Picture 11

delay before the processor senses that the interrupt request is turned off. To ensure sufficient time for the request to drop, eight additional cycles should intervene between the last "IOSTROBE" (setting of bit M2.IOS to effect the I/O transfer) and a query of the interrupt system.

5.4.2 Interrupt Enable Register

The microcode can also disable the interrupt requests from the two USARTs with a mask in a 4-bit Interrupt Enable Register (IER). The IER holds four interrupt mask bits, corresponding to the transmit and receive interrupts from each USART:

bit	interrupt
0	receive, device 0 (console)
1	transmit, device 0 (console)
2	receive, device 1 (loader)
3	transmit, device 1 (loader)

Setting a bit enables the interrupt; clearing a bit disables the interrupt. The low order 4-bits of the IER are read/write bits; the high order 4 bits are always read as 0's and cannot be written. (The current system software does not use the IER's mask bits to disable interrupts.)

5.4.3 Switch Register

The fourth I/O device is an 8-bit switch register located on the MBB's memory board. The contents of these switches may be read from device 6 on board 0 (see figure 5.2). Writing to this location does nothing. The low 4 switches are used by the system software to select the console terminal's baud rate. The high 4 switches are currently unused.

5.5 Interrupt System

It is important not to confuse the microinterrupt system with the mechanism for handling exceptional conditions (also referred to as "traps"). Exceptional conditions are discussed in Section 2.12.

Microinterrupts, such as those for device transfer completions, are queued by the interrupt system as service requests. Such requests are serviced only when polled by the microprogram. The hardware supplies a "vector address" corresponding to the highest priority pending request. (Transferring control to that address would service the request.) If no interrupt request is pending, the hardware supplies an address "MAIN" corresponding to the microprogram's main task. This task is usually used to emulate macroinstructions; transferring control to "MAIN" would cause the next macroinstruction to be emulated. To dismiss and transfer control to the handler for the highest priority pending request (or to "MAIN" if no requests are pending), the microcode loads the hardware-supplied vector address into the UPC. Interrupt vectors are located in RAM code, so that service routines may be changed easily.

Microcode routines are thus free from being interrupted (except by traps). Code is divided into strips, each of which dismisses by loading the interrupt vector into the UPC. Strips must be short enough to provide response to I/O interrupts within the latency constraints of the various interfaces.

5.5.1 Device Priority and Vectors

The priority of interrupt requests is governed by an interrupt chain which begins on the memory board, goes "up" the I/O boards (from lowest numbered board to highest), comes "down" the I/O boards, and finishes on the memory board. Thus any I/O board may have both "high priority" devices, by inserting them in the "up" chain, and/or "low priority" devices, by inserting them in the "down" chain. The last I/O board must have a jumper cable to connect the "up" and "down" interrupt chains.

The MBB's standard I/O devices include both high priority and low priority devices, and, in fact, since the chain begins and ends on this board they are respectively the highest and lowest priority interrupts. Besides interrupts from the above-mentioned communications interfaces, standard interrupts include one which is generated every 100 microseconds and a programmable interrupt.

The 100 microsecond interrupt is used, among other things, to cause the refresh of main memory and to drive a timing "service", which allows users' microcode to be called at specified intervals. The programmable interrupt can be set and cleared by the microprogram. It has the lowest priority save that of "MAIN". Thus it is useful for tasks which should proceed before the next macroinstruction but should yield to I/O service. One example is tracing of macroinstructions, in which some state is recorded before each macroinstruction is executed. Another use is to allow long microprogram strips to be broken up to meet I/O latency requirements.

Each interrupt has an associated vector. Vectors are located at the start of the upper 8K (the loadable half) of microcode. Vectors are spaced 4 addresses apart, to permit context saving and branching. Vector addresses have eight bits of significance, three to denote the I/O board of their origin and five to indicate the device within the I/O board (see Figure 5.3). Vector addresses thus range from 20000 to 21774.

INTERRUPT VECTOR ADDRESS

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	BOARD			DEVICE					Ø	Ø

Figure 5.3 I/O Interrupt Vector Addresses
Picture 12

If an I/O board has fewer than 32 devices or there are fewer than seven I/O boards in a system, some of the microcode address space in the reserved vector area will not be used for vectors. This space is then available for any other microcode.

Interrupt priority, from highest to lowest, along with the vector addresses of the core devices, is as follows:

vector address (octal)	
20004	100 microsecond clock
20020	console interface, receiving side
20024	console interface, sending side
20030	loader interface, receiving side
20034	loader interface, sending side
	<other devices in the high priority chain>
	<other devices in the low priority chain>
20040	microprogrammable interrupt
20050	MAIN (instruction emulation)

5.5.2 Servicing Interrupt Requests

The interrupt vector source (called "INTS") gives the vector of the highest requested interrupt. Since loading the interrupt vector into the UPC transfers control to the vector, the microcode can easily dismiss to permit priority-ordered interrupt service. The interrupt system works by polled requests rather than actual interrupts so there is no need to "lock out" interrupts. The "MAIN" request is always asserted; its vector should branch to the machine's background level task, such as emulating the next macroinstruction.

We recommend reserving 20000 as an illegal vector, since various hardware and software malfunctions can transfer control there. Two such observed malfunctions have been broken hardware in the interrupt chain and improper dispatching (Section 2.8) through a cleared Dispatch memory cell.

Servicing a request does not automatically clear the request; the microcode handler must explicitly clear it. The clock sets its request every 100 microseconds; writing a 1 to the M2.CLKINT bit in MISC2 clears the request. (When read, M2.CLKINT reflects the clock interrupt request state: 1 for on, 0 for off.) The console and loader interfaces assert requests when ready to supply received data or accept data for transmission; the requests are turned off by reading or writing data, or by disabling transmission or reception. (These requests can also be disabled by mask bits in the Interrupt Enable Register.) The programmable interrupt is requested by the M.PROGINT bit in MISC. Setting the bit sets the request; clearing the bit clears the request.

5.5.3 Interrupt Timing Constraints

There is a delay between events which set or clear requests and the reflection of the new state in the INTS source. The interrupt hardware periodically samples the states of the various interrupt service request lines, and places on the interrupt bus the vector address corresponding to the highest priority device requesting service. Because of timing considerations in determining the highest priority request, the interrupt system is controlled by a special clock signal which runs at half the speed of the main MBB system clock; interrupt requests are thus sampled every two microcycles.

To be sure of updated information, some number of microcycles must intervene between the program's setting or clearing a request and the program's sampling the INTS source. This number of cycles is dependent on each I/O board's implementation. For the core devices, four cycles suffice for the 100 microsecond clock and the programmable interrupt. After clearing a console or loader interface interrupt by reading, writing, or disabling transmission or reception, a wait of eight cycles (instead of four) should intervene between the last IOSTROBE cycle and sampling the INTS source.

6 MICROINSTRUCTION FORMAT

Figure 6.1 shows the microinstruction format. We now explain each field in detail. The decimal bit numbers of each field and subfield are given below in parentheses. Field and subfield values are given in hexadecimal. The mnemonics listed are known to the assembler and the Control and Debugging Package.

The fields and subfields which control writing (Latch Control, Map Control, Register Load Control, Destination Type, and Destination Register) are interpreted so that a value of 0 avoids writing anywhere. Thus, a microinstruction with value 1 (all bits 0 except for the Parity bit to maintain odd parity) acts as a "no-op".

6.1 Source Field (UIR 31 - 20)

The Source field controls driving of the source bus. (It also controls whether to drive the destination bus from the ALU output or from Dispatch memory; see code 4 below.) The source bus is 20 bits wide. The Source Type subfield (31-28) makes the primary selection as follows:

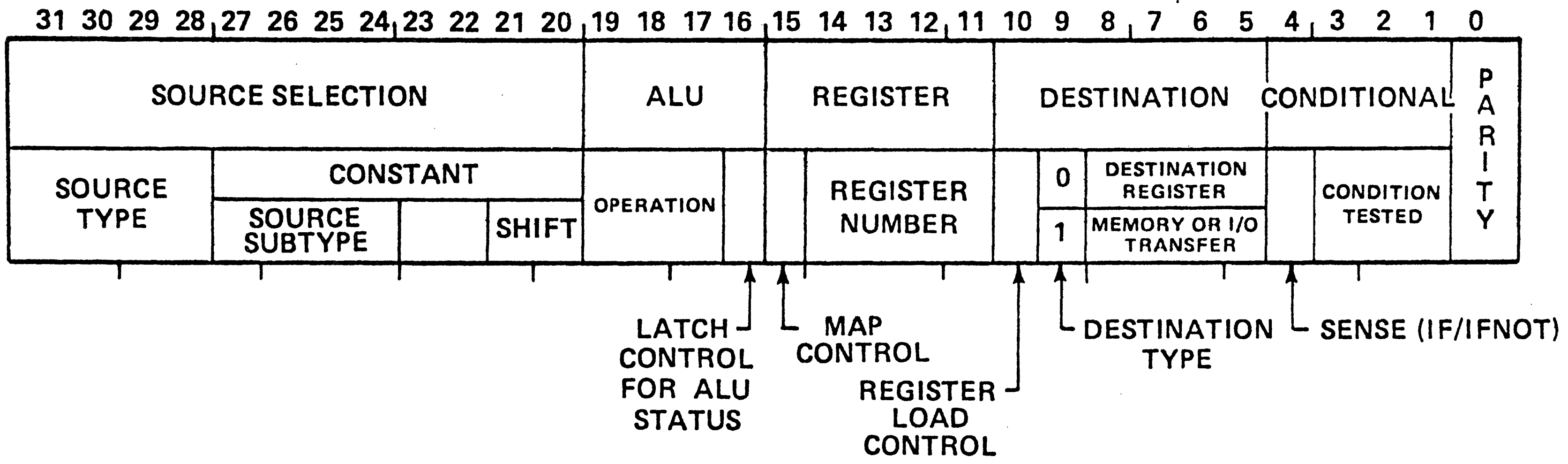


Figure 6.1: Microinstruction Format
Picture 13

Source Type	
0-3	Use a source given in the Source Subtype field. Pad the high order bits of the source bus with 0's for sources less than 20 bits according to width of source as follows:
4	Place 0 on the source bus, but drive the destination bus from Dispatch memory instead of from the ALU output. The assembler mnemonic for this is DISP. (The Constant subfield may be used by the Dispatch address selection logic on the MIRDB.)

Padding for source types 0-3:

0	don't pad	(20 bit source)
1	pad top 4 bits	(16 bit source)
2	pad top 8 bits	(12 bit source)
3	pad top 12 bits	(8 bit source)

5	Place 0 on the source bus. (The assembler never uses this code.)
6-9	Use an "extended constant" of 14 bits. Drive the source bus's bottom 8 bits from the Constant subfield; the next 4 bits from the ALU field; the next 2 bits from the Source Type code:

6	10
7	11
8	00
9	01

The top 6 bits will be 0's. The ALU is forced to perform a "pass" operation, and ALU status will not be latched.

A-F Use a "regular constant". The source bus will have 8 contiguous bits driven from the Constant subfield, with the other bits either all 0's or all 1's, determined by the Source Type code as follows:

code	placement	padding
A	15-8	1's
B	15-8	0's
C	19-12	1's
D	19-12	0's
E	7-0	1's
F	7-0	0's

The following table summarizes the effect of the Source Type subfield:

Source Bus

Type Code	Bits 19-16	Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0	use
0	*	*	*	*	*	20-bit source
1	0	*	*	*	*	16-bit source
2	0	0	*	*	*	12-bit source
3	0	0	0	*	*	8-bit source
4	0	0	0	0	0	Dispatch
5	0	0	0	0	0	(unused)
6	0	"2"	ALU	M	L	extended constant
7	0	"3"	ALU	M	L	extended constant
8	0	"0"	ALU	M	L	extended constant
9	0	"1"	ALU	M	L	extended constant
A	1's	M	L	1's	1's	regular constant
B	0	M	L	0	0	regular constant
C	M	L	1's	1's	1's	regular constant
D	M	L	0	0	0	regular constant
E	1's	1's	1's	M	L	regular constant
F	0	0	0	M	L	regular constant

* => source specified in Source Subtype subfield
ALU => ALU field (bits 19-16)
L => low 4 bits, source field (bits 23-20)
M => middle 4 bits, source field (bits 27-24)

The Source Subtype (bits 27-24) specifies source as follows:

Source Type Code	Source Subtype Code	Source	Mnemonic	Source Width
0	0	Memory Address Register	MAR	20
0	1	Memory Buffer Register	MBR	20
0	2	Temp	TEMP	20
1	3	Interrupt Vector	INTS	16
3	4	ALU Status	ALUST	08
1	5	Serial Number	SN	16
1	6	MISC	MISC	16
1	7	MISC2	MISC2	16
1	8	EDAC	EDAC	16
1	9	Micro-RAM, low bits	URAML	16
1	A	Micro-RAM, high bits	URAMH	16
0	B	Macroinstruction Register	MIR	20
0	C	Macroinstruction Field	MIRFLD	20
0	D	Swapped MBR (8 bit bytes)	S16MBR	20
2	E	Base	BASE	16
0	F	Swapped MBR (10 bit bytes)	S20MBR	20

The Source Type code must specify padding appropriate to the source's width. (The assembler assembles the appropriate code automatically.) The source's contents are right justified on the source bus, and the padding of 0's drives the high order bits. If padding is improperly specified, so that some bits on the source bus are driven by both the padding and the source, the hardware may be damaged.

If the MAR is specified as the source, bits 21 and 20 specify the MAR/MBR shift, if any:

UIR	Action	Mnemonic
0	no shift	MAR
1	shift right	MAR (SR)
2	shift left	MAR (SL)
3	undefined	(none)

6.2 ALU Field (UIR 19 - 16)

The ALU field controls the ALU. The Operation subfield (19 -17) specifies an ALU operation as follows:

Code	Operation	Mnemonic
0	AND	&
1	ADD	+
2	INclusive OR	!
3	PASS bus	
4	Complement bus	NOT
5	(undefined)	
6	EXclusive OR	?
7	SUBtract (register minus bus)	-

If the Latch Control bit (16) is on, the ALU status is latched into the ALU status register; if this bit is off, the ALU status is not latched.

If the Source field calls for an extended constant, the ALU automatically does a pass operation; the ALU field is then used to help specify the constant. Note that the ALU status cannot be latched during an extended constant instruction.

6.3 Register Field (UIR 15 - 11)

The Register field specifies the ALU scratch register input to the ALU. If the Map Control bit (15) is on, the Register Number subfield (14-11) gives the register address. (If, however, the Register field has value 30 or 31, then the MIR daughterboard supplies a four-bit register number which, when inclusive ORed with the contents of BASE, gives the register address.) If the Map Control bit is off, the Register Number subfield, inclusive ORed with the contents of BASE, gives the register address.

6.4 Destination Field (UIR 10 - 5)

The Destination field specifies which register is loaded from the destination bus. (The Destination bus itself is driven by the ALU output unless dispatch memory is used as a source, in which case the bus is driven directly from Dispatch memory.) If a destination register has fewer than 20 bits, the high order bits of the bus are not used.

If the Register Load Control bit (10) is on, the ALU result is written to the scratch register specified in the Register field. (Thus, the same register that was used as the ALU's "A" input is written.)

If the Destination Type bit (9) is off, the Destination Register field (8-5) selects a destination as follows:

Code	Destination	Mnemonic
0	(none)	
1	MBR	MBR
2	TEMP	TEMP
3	MIR	MIR
4	RAMADDR	RAMADDR
5	UPC and RAMADDR	UPC
6	MISC	MISC
7	MISC2	MISC2
8	EDAC control	EDAC
9	ALU status	ALUST
A	BASE	BASE

If the Destination Type bit is on, the Destination is the MAR. The Transfer subfield (8-5) then specifies the memory or I/O operation (MAROP), if any. Defined codes are as follows:

Code	Transfer	Mnemonic
0	None (just load MAR)	
2	write I/O	WIO
3	read I/O	RIO
A	read physical	RP
B	write physical	WP
C	read macro	R
D	write macro	W

(The low order bit is a "read/write" bit. For memory operations, 0 means read and 1 means write. For I/O operations, 0 means write and 1 means read.)

6.5 Condition Field (UIR 4-1)

The Condition field specifies conditional execution. The Sense bit (4) gives the sense of the condition. A 0 means to execute the instruction only if the condition is true; a 1 means to execute the instruction only if the condition is false. The Condition Tested subfield specifies a condition as follows:

Code	Condition	Mnemonic
0	(always true)	TRUE
1	ALU result zero	ZERO
2	ALU result negative	NEG
3	ALU result odd	ODD
4	MAR condition (from the MARDB)	MARCOND
5	interrupt pending	INTP
6	mode flag 0 is on	MODEF0
7	(always false, not used)	

The ALU conditions refer to the appropriate bits in the ALU Status Register. The MAR condition is controlled by the MAR daughterboard. The "interrupt pending" condition is true if any microinterrupt request other than the software-controlled request is pending. Mode flag 0, a bit in MISC, is controlled by the microprogram.

6.6 Parity (UIR 0)

The Parity bit gives the instruction odd parity.

6.7 Special Considerations for Using Dispatch as a Source

When Dispatch memory is used as a source (Source Type code 4), the destination bus is driven by Dispatch memory instead of by the ALU output. The ALU operation subfield is not useful; and if the Latch Control bit is on, the ALU status register is latched with random values. The Register field can specify a scratch register to be driven from the destination bus (the Load Register bit must be on as usual), and the Destination field selects a destination as usual.

7 BIT ASSIGNMENTS IN CERTAIN REGISTERS

We will discuss the bit assignments for various registers: ALUST, MISC, MISC2, and EDAC. As in Section 6, the mnemonics listed are known to the assembler and the Control and Debugging Package. The symbols' values are masks for the indicated bits or fields.

7.1 ALUST

The ALUST register may be loaded directly from the destination bus. Normally, however, it is used to record the ALU status. When the ALU status is latched, the ALU status register is loaded as follows:

Bit	Meaning
0	{on if ALU result is zero}
1	low bit of ALU result
2	(always latched as 0)
3	(always latched as 0)
4	ALU "carry"
5	high bit of ALU result
6	high bit of source bus input to the ALU
7	high bit of scratch register input to the ALU

The "high bit" is bit 15 or 19, depending on whether the machine is in 16-bit or 20-bit data width mode. For determining whether the ALU result is zero, the high four bits are ignored in 16-bit mode. Bits 0, 1, and 5 of ALUST determine the conditions ZERO, ODD, and NEG respectively for conditional execution. The ALU "carry" is defined in Section 2.3.

7.2 MISC

The MISC register controls various micromachine functions. Like most MBB registers, MISC "holds" bit values: bits' values written may be read back intact. The bits are assigned as follows:

Bit	Function	Mnemonic
0	mode flag 0	M.MODEF0
1	mode flag 1	M.MODEF1
2	mode flag 2	M.MODEF2
3	mode flag 3	M.MODEF3
4	programmable interrupt	M.PROGINT
5	Data Width Mode	M.DWMODE
6	Data Width Control	M.DWCTRL
7	Load Dispatch	M.LOADDISP
8	Light 0	M.LITE0
9	Light 1	M.LITE1
10	Light 2	M.LITE2
11	Light 3	M.LITE3

Mode flag 0 is one of the testable conditions for conditional execution. The four mode flags have no pre-assigned meaning in the MBB; they are for use by the application-dependent MIR and MAR daughterboards. (For the H316 emulation, for example, flag 0 on indicates "extended" addressing mode, and flag 1 on indicates "upper bank" when in "normal" addressing mode. The MIR daughterboard uses flag 0, and the MAR daughterboard uses both flags.) When M.PROGINT is on, the programmable interrupt is requested; the request must be explicitly cleared by resetting the M.PROGINT bit.

The MISC register controls the data width used for shifting and determining ALU status conditions. If M.DWCTRL is on, the machine's data width mode is controlled by the MIR daughterboard. If M.DWCTRL is off, the machine's data width mode is controlled by the M.DWMODE of MISC: 0 for 16 bits, 1 for 20 bits.

M.LOADDISP must be set to load Dispatch memory (Section 3.6). The lights bits drive LEDs on the memory board which are visible from the front of the MBB. A 1 corresponds to the light on. When looking at the front panel, light 0 is on the right.

7.3 MISC2

MISC2 and the EDAC register, unlike most registers, do not "hold" written values in the conventional sense; their "read" and "write" values need not be obviously related. Here are the bits in MISC2:

Bit	Write	Read	Mnemonic
0	(clear)	new power	M2.NEWPWR
1	(clear)	microcode parity error	M2.UPERR
2	IORESET	button being pushed	M2.BUTTON M2.IORESET
3	IOSTROBE	0	M2.IOS
4	load uRAM low	0	M2.LOADUL
5	load uRAM high	0	M2.LOADUH
6	--	AC power ok	M2.ACOK
7	--	battery fully charged	M2.BATOK
8	(clear)	any main memory error	M2.MEMERR
9	(clear)	uncorrectable main memory error	M2.UCERR
10	(clear)	clock interrupt	M2.CLKINT
11	--	MBR parity	M2.MBRPAR
12-15	--	MAROP0 - MAROP3	M2.MAROP

M2.NEWPWR, M2.UPERR, M2.MEMERR, and M2.UCERR behave in a "write-to-clear" manner. Normally, these bits are read as 0. After the associated condition occurs, the bit will be read as 1. Writing a 1 to the bit in question clears the condition indication; the bit will then be read again as 0 until the condition next occurs. (When the condition indication is off, writing a 1 to the bit does nothing.)

The clock interrupt bit behaves the same way, but this bit also controls the interrupt request. Every 100 microseconds, the clock asserts its interrupt request and sets the "read" side of M2.CLKINT. Both the interrupt request and the "read" side of M2.CLKINT will stay on until a 1 is written to M2.CLKINT. (Writing a 1 when the request is off does nothing.)

M2.IORESET and M2.BUTTON share a single bit in MISC2. M2.IORESET is used to reset I/O devices. When this bit is set, the IOVALID signal is not asserted, and all I/O devices are reset, except the standard I/O devices discussed in Section 5.4. M2.BUTTON is off unless the button is currently being pushed.

M2.IOS controls the IOSTROBE signal, which is used to initiate I/O transfers. This signal is described in Section 5.2.

Writing a 1 to M2.LOADUL or M2.LOADUH loads microcode RAM. TEMP contains the contents, and RAMADDR contains the address; M2.LOADUL loads the low 16 bits in the specified address, and M2.LOADUH loads the high 16 bits. Section 2.10 explains further.

M2.ACOK is on when the AC line voltage is within acceptable operating limits for the MBB's power supply, and off when the battery is being used to supplement inadequate AC power. M2.BATOK is on whenever the battery is fully charged.

M2.MBRPAR gives the odd parity of the MBR. If the EDAC option is installed, this parity includes the MBR's check bits. (After an explicit load of the MBR from the destination bus, all six check bits are normally set to 1; so the check bits should not alter the parity.)

M2.MAROP reflects the machine's MAROP field, which is a copy of the Transfer field (bits 8-5) of the microinstruction which last loaded the MAR.

7.4 The EDAC Register

Like MISC2, the EDAC register has independent "read" and "write" sides. This register deals with the optional main memory Error Detection and Correction (EDAC) logic described in Section 4. The field assignments are:

Bits	Write	Read	Mnemonic
0-5	check bit values to load	read check pattern on last error	EDAC.PREP
6	enable fixing	0	EDAC.FIX
7	enable traps	0	EDAC.TRAP
8-11	--	memory bank	EDAC.BANK
12-15	--	MAR type from MARD	EDAC.MAR

Writing EDAC.PREP sets the values which are loaded into the MBR's check bits when the MBR is loaded from the destination bus. (This field must be set to all 1's for normal operation.) This field's "read" side gives the read check pattern latched on the last correctable error when fixing was enabled. (Between power-up and the first correctable error, the value is random.) The EDAC.FIX bit being on enables correction of single bit errors. The EDAC.TRAP bit enables trapping to microcode address 0 on double (uncorrectable) errors. When a single error occurs with fixing enabled, the number of the physical memory bank being referenced is latched in EDAC.BANK. The EDAC.PREP and EDAC.BANK fields together identify the individual memory chip which caused the error (see Chapter 4). Application-dependent information from the MAR daughterboard can be referenced via the 4 bit

EDAC.MAR field.

8 THE MBB's SYSTEM SOFTWARE

This section describes the MBB's current system software. The system uses the two terminal interfaces to support loading, DDT, and application-dependent traffic; handles exceptional conditions; and otherwise provides a friendly environment for application software. The program listing (pointed to in Section 8.4) gives details omitted here.

8.1 The Terminal Handlers

The system software centers around two terminal handlers. They are independent twin processes, sharing reentrant code and using different data areas in the ALU register file.

Each terminal handler independently multiplexes its traffic among three routines: a micro-DDT (UDDT), a loader, and the application software. UDDT receives and responds to commands from the external device. (Section 9 discusses UDDT commands.) The loader loads the MBB's memories from an external device. The application software may use the terminal interface for application-dependent purposes. The terminal handler switches control among these routines based on commands received from the external device; only one routine controls the interface at a time. The "terminal handler" denotes the hardware server plus these three routines behind the server.

In principle, the MBB can be completely controlled through either of its two terminal interfaces. In fact, however, the kind of control exercised may be limited by the nature of the external device. Generally, a terminal might not be suited for loading and a cassette might not be suited for UDDT. A terminal with cassette, or another computer, could be suited for both UDDT and loading.

The two terminal handlers are truly independent. For example, one handler can support application traffic while the other handles UDDT commands. Or both handlers can perform loading at once; if they both want to load the same area, the second will overwrite what the first has previously written.

The UDDT and loader routines are partly in PROM and partly in RAM. Until the RAM portion is loaded, some UDDT commands are unavailable.

8.2 'Running' and 'Not Running' States

The system software distinguishes two states of the MBB: "running" the application software (such as macrocode emulation) and "not running". When "not running", the microcode executes a tight loop with three coroutines: one routine which refreshes main memory, plus the two terminal handlers. Each terminal handler sleeps (returns) when waiting for character input or output. Thus, the interfaces are continually polled. The microinterrupt system is not used (that is, the instruction "INTS->UPC" is never executed).

Before the MBB enters the "running" state, the RAM-resident system software must be loaded. The "G" command to UDDT sets the running state. (UDDT sets the system's "running" flag and calls the application program's initialization routine.) When the MBB is "running", program flow is different. All routines regularly dismiss with the microinstruction "INTS->UPC", and interrupts (serviced by system and application code in RAM) drive the MBB's processes. When any input or output interrupt occurs on either standard interface, both terminal handlers are awakened. (Awakening both handlers, though not necessary, is the easiest thing to do.) As before, each terminal handler sleeps (returns) when waiting for I/O. After both handlers return, an "INTS->UPC" is executed, dismissing until the next interrupt. The clock interrupt handler takes over the refreshing of main memory. (The clock interrupt handler also offers a wake-up service for the application program.) The system software services only the clock interrupt and the terminal I/O interrupts; the application software services MAIN, the programmable interrupt (which it controls), and any interrupts for application-dependent I/O.

The MBB stays in the "running" state until UDDT processes a halt command ("H") or an exceptional condition causes the system to reinitialize.

8.3 Exceptional Conditions and Initialization

Various exceptional conditions cause the system to reinitialize. On sensing such a condition, the system stops running application code, saves the readable registers (such as MAR and TEMP) in a block of ALU scratch registers, initializes both terminal handlers, and sends out through both interfaces a character code for the type of error. The condition may be hardware-detected, trapping through microcode address 0, or software-detected, branching through a UDDT breakpoint routine.

The character codes for the different conditions are:

n	New power
b	Button pushed
p	Parity error in microcode
m	uncorrectable Main Memory error
j	wild Jump to address 0 (no apparent reason)
*	microcode breakpoint
t	macrocode trap or breakpoint

All except "*" and "t" indicate hardware-detected conditions which trapped to address 0.

Certain of these errors necessitate some extra initialization beyond that described above. On new power, all non-PROM memories (microcode RAM, Dispatch, main memory, ALU scratch registers) are cleared. When the button is pushed, before letting the terminal handlers run, the microcode stays in a tight loop (which includes a main memory refresh) until the button is released. (One reason for this wait is to avoid suppression of traps. As explained in Section 2.12, as long as the button is pushed, microcode parity errors and main memory errors cannot cause traps.) When the loader port has a boot device attached, the MBB will automatically load itself on all exception conditions except button pushed and microbreakpoint.

Unfortunately, when an exceptional condition occurs, the UPC cannot be saved for inspection.

8.4 Source File

The system software source is in two files: a PROM resident section and a URAM resident section. The files currently reside in <MBB> as USYS-PROM.MIC and USYS-URAM.MIC.

9 THE MBB's UDDT COMMANDS

This section discusses the micro-DDT (UDDT) commands, as well as one command to transfer control from the application software to UDDT. This section assumes familiarity with DDTs in general and does not precisely describe all formats, conventions, and shortcuts. The letter commands given below must be entered in upper case. In this section, "n" represents an octal number.

The MBB prompts with carriage return and line feed followed by "!".

The commands "nU", "nM", "nD", and "nR" examine address "n" in microcode memory, main memory, Dispatch memory, and the ALU scratch registers respectively. An MIR daughterboard must be installed before Dispatch memory can be accessed. The MBB responds to these commands with the value in the location specified. The lower 8K of microcode memory may not be accessed. Main memory is accessed in the "physical" mode; similarly, Dispatch memory and the registers are accessed by absolute addresses without mapping. (Dispatch is interrogated with the microinstruction constant field containing 0; the MIR daughterboard is then presumed to effect a "transparent" map, as described in Section 2.8.)

Terminating characters act as with many other DDTs. A carriage return closes the current location; a line feed closes the current one and opens the next; and an up-arrow closes the current one and opens the previous one. A number, if entered before the terminating character, is stored as the new value in the open location.

Commands "nJ", "nG", and "H" control program execution. "nJ" transfers microcode execution to address "n". This command is useful for branching to hardware test programs. The code called by the "J" command may return to the system software by branching to "pdt.int"; in this case, UDDT will still control the terminal interface. (To run a test program repeatedly in parallel with the system software, one can put the test program at the MAIN interrupt vector. The test program is then the "application" program, and it may be started by the "G" command discussed in the next paragraph. The test program should dismiss with "INTS->UPC".)

"nG" starts the application software. The argument "n" is used by the application software, perhaps as a macrocode starting address; the argument may also be missing, in which case the application software is so informed. "H" halts the application software.

On receiving the command "E", UDDT releases control of the interface to the application software. Further incoming characters (except control-N and control-E) are then passed to the application software, and the application software may then send characters to the external device. "E" is legal only if the application software is running. A control-N returns terminal control to UDDT.

Control-E switches the character echoing state. (Control-E has this effect whether UDDT or the application software is in control.) Initially, the terminal handler echoes received characters. A control-E inhibits echoing; a second control-E reenables echoing; and so on.

Figure 9.1 summarizes the transfer of control among UDDT, the loader, and the application software.

Rubout echoes "X" and makes UDDT forget the latest typein. Spaces are ignored, and input characters not yet discussed is illegal. On receiving an illegal input, the UDDT closes any open location and types "?" followed by a prompt.

Table 9.1 summarizes the commands. Certain commands' handlers are in RAM; these commands, marked in the table by '*', are allowed only when UDDT's RAM extension is loaded.

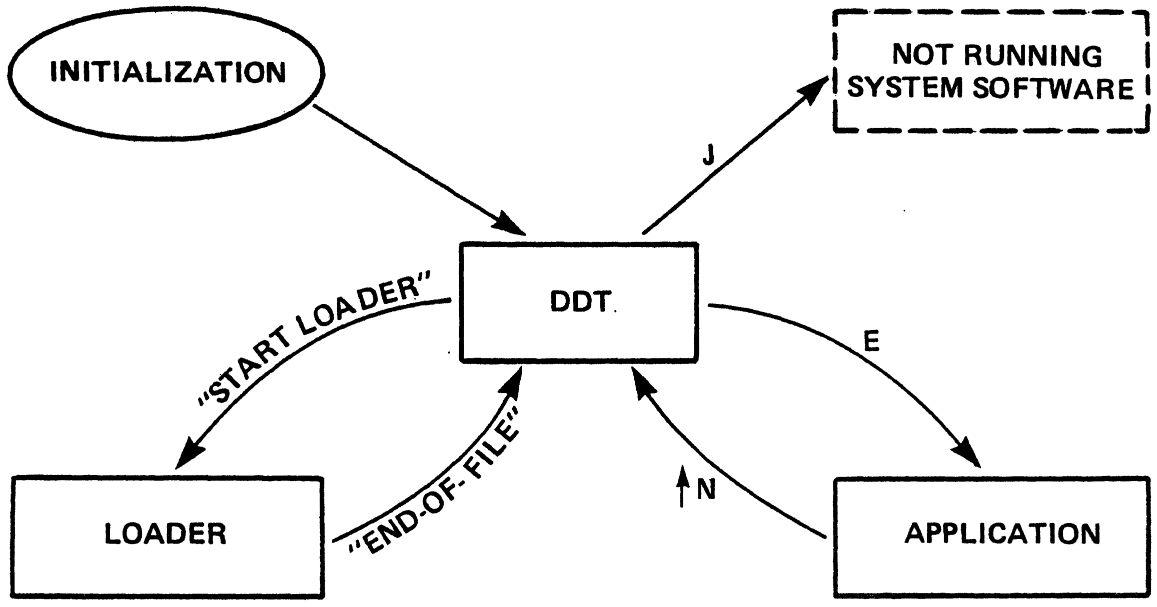


Figure 9.1: Control of the Terminal Interface
Picture 14

nU	examine microcode memory
nM	examine Main memory
nD	examine Dispatch memory
nR	examine Registers
(n)<carriage return>	close location (inserting new value if supplied)
(n)<line feed>	close location (inserting new value if supplied) and open next one
*(n) <uparrow>	close location (inserting new value if supplied) and open previous one
nJ	Jump to microcode address
*(n)G	Start application software (Go)
*H	Halt application software
*E	Exit DDT (connect the terminal to the application program)
control-N	return terminal control to UDDT
control-E	change Echo state
<rubout>	close any open location, and echo "X"
<space>	ignored

*=>allowed only if UDDT RAM microcode is loaded

( ) indicates an optional argument

Table 9.1: The MBB's UDDT Commands

10 THE CONTROL AND DEBUGGING PACKAGE

The Control and Debugging Package (CDP) runs on TENEX and TOPS-20. It communicates with the MBB's system software (Section 8) through a PTIP port connected to one of the MBB's terminal interfaces. The CDP user can issue DDT commands to the MBB either in an uninterpreted manner or in an interpreted (symbolic) manner. The user can also load the MBB from a local file. Further, the user can activate a simulated MBB with special features such as breakpoints; the user interacts with the simulated MBB in much the same manner as he interacts with the real MBB.

Lower case letters typed to the CDP are converted to upper case on input. In this section, "$" denotes "escape".

We will shortly describe the CDP in some detail, considering the CDP's internal states and the associated internal states of the MBB's terminal handler. The CDP functions in a complicated manner, so this description will be complicated. For the naive user, however, the CDP can be easy to use; some of the CDP's internal complexity even stems from the effort to ease the naive user's way. Some key commands work as follows: "$G" starts the application microcode. Control-N returns the user's terminal to the top level (just as it does with the MBB's UDDT). Control-X halts whatever can be halted, and returns the user's terminal to top level. "$P" proceeds from any interruption, undoing a previous control-N or control-X.

Whether interacting with the real or simulated MBB, the CDP hides complexity from the user. A single command from the user may cause several interactions between the CDP and the MBB, including unobvious items such as echo control.

In studying the CDP's functioning, it is easier at first to consider interactions with the real MBB only. Much of what we say will apply also to the simulated MBB; later we will treat the similarities and differences.

10.1 The CDP's Modes

The CDP has four modes or levels: command, loading, application, and system. The first three modes correspond to the controller of the MBB's terminal handler (Section 8.1). In command mode, the CDP talks to the MBB's UDDT; the CDP translates between the user's symbolic command language (described below) and UDDT's command language (descried in Section 9). In loading mode, the CDP talks to the MBB's loader in order to load a file. In application mode, the CDP talks to the MBB's application software; the CDP cooperates with the MBB's terminal handler to pass characters directly between the user and the MBB's application software. In system mode, as in command mode, the CDP talks with UDDT; in this case, however, the CDP passes characters without interpretation between the user and UDDT. The user may thus use directly the MBB's UDDT commands discussed in Section 9.

We now describe the rather tricky rules for switching between modes, summarized in Figure 10.1.

Certain events always cause the CDP to enter command mode. On receiving notice from the MBB of an exceptional condition, the CDP prints an appropriate message and enters command mode. As discussed in Section 10.3, the MBB sends a character code for each exceptional condition. The messages typed for each code are as follows:

character received from MBB	notice typed to user
n	Power on
b	Button
p	Microcode parity error
m	Main memory error
j	Jumped to zero
*	Program aborted
t	Instruction trap

If the user at any time types five control-X characters in a row, the CDP does a "panic" return to command level. This panic return may leave things in a funny state. For example, if the panic return skipped the step of turning off echoes from the MBB, the CDP will be confused by unexpected incoming echoes.

At command level, ";L" tells the CDP to enter loading mode. After the user specifies a file, the CDP uses UDDT commands to

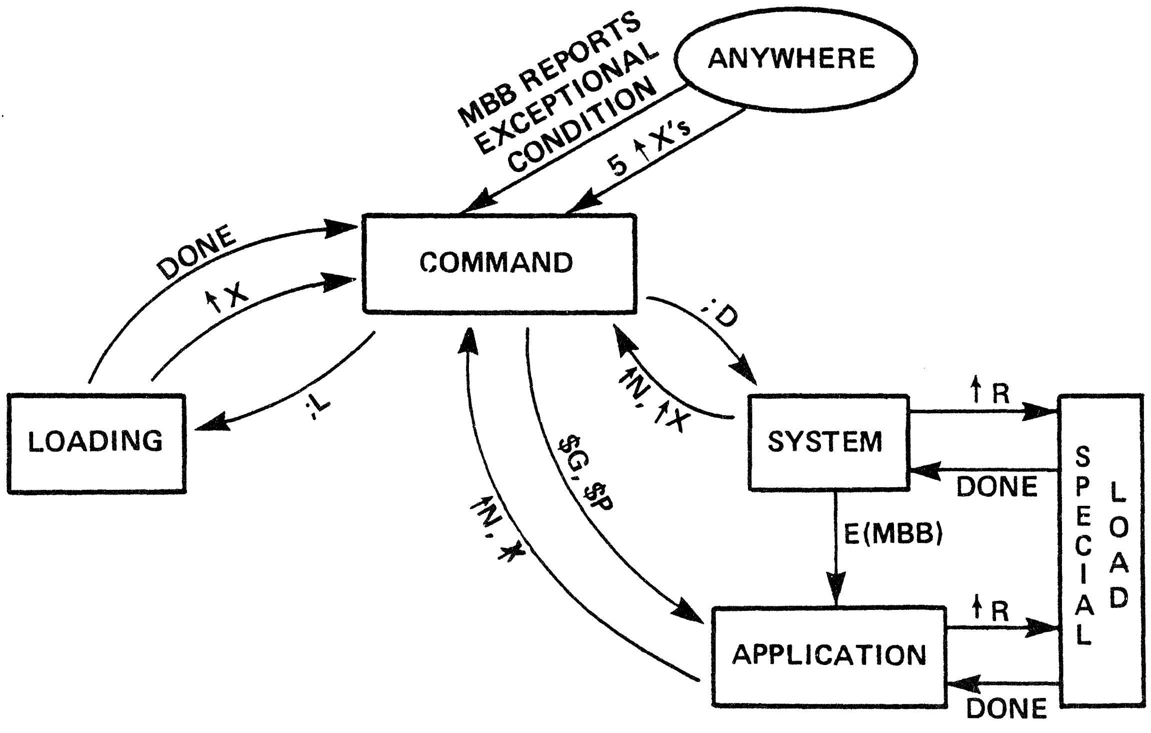


Figure 10.1: The CDP's States
Picture 15

load a small bootstrap. When loaded, the bootstrap starts running and loads in the real loader. The real loader automatically starts running after being loaded. The CDP then sends the file to the real loader. When done, the CDP sends the loader an "end-of-file" block, causing the loader to release control back to UDDT. The CDP then reenters command level to accept further commands. During a load, control-X tells the CDP to quit. The CDP immediately sends the MBB's loader an "end-of-file" block, causing it to release control to UDDT, and the CDP reenters to command level.

Most commonly, the CDP is in either command or application mode. The CDP starts out in command mode. From command mode, "$G" and "$P" enter application mode. (These commands also have other effects described below.) In application mode, control-N and control-X reenter command mode. (Control-X also has other effects described below; control-N has no other effect.)

System mode is normally used only to debug the MBB's system code. At command level, ";D" enters system mode. Actually, system and application modes are not distinguished by the CDP. In both modes, characters are passed transparently between the user and the MBB; the only difference is in the MBB itself. We say that the CDP is in "system" mode if it is talking with UDDT while we say that the CDP is in "application" mode if it is talking with the MBB's application software.

If the CDP is in system mode and the user types "E", UDDT, on receiving the character, will release control to the application software; the CDP is now in application mode! Even though a control-N, if received at the MBB, would restore control of the MBB's interface to UDDT, the CDP user cannot return to system mode simply by typing control-N. Indeed, control-N is not sent to the MBB but intercepted by the CDP as a command to return to command mode. Following the control-N with ";D" will reenter system mode.

In system and application modes, a special feature is available by typing control-R. After the user specifies a file, the CDP simply sends the raw file to the MBB. This feature permits sending a "paper tape" file to the application software, and it generally would be used only in application mode. After the file has been sent, the user is still in system or application mode.

10.2 The CDP's Command Mode

This section describes the commands available within the command level, and also those which return to command level from application mode. Many of these commands cause the CDP to interact with the MBB's terminal handler. With some of the commands, we will sketch this interaction. These sketches will ignore subtle timing considerations, such as the need for the CDP to suppress MBB echoing or to wait for a response from the MBB.

10.2.1 Examining and Changing Locations

To examine and change locations is straightforward. Where n is a number, "n U/", "n M/", "n D/", and "n R/" examine location n in microcode memory, Main memory, Dispatch memory, and the ALU scratch registers respectively. (The CDP sends UDDT an "nU", "nM", "nD", or "nR" command.) The CDP remembers a "current address space", which is one of these four memories. Examining a location in a given memory sets the current address space to that memory. The command ";C" followed by "U", "M", "D", or "R" also sets the current memory space. Typing "n/" examines location n in the current space.

Terminating characters act as with many other DDTs. A carriage return closes the current location; a line feed closes the current one and opens the next; and an up-arrow closes the current one and opens the previous one. A number, if entered before the terminating characters, is stored as the new value in the open location.

10.2.2 Symbols

Numbers need not be octal. The prevailing radix may be set to any value m, 2<m<16, by "m$R". (m is interpreted as a decimal number.) Initially, 8 ("octal") is used.

On startup, the CDP loads the MBB assembler's symbols; and whenever the CDP loads a file into the MBB it retains that file's symbols. (MBB binary files prepared by the assembler include a symbol table.) In general, one may use symbols whenever one could use numbers. One may even use well formed arithmetic expressions; expressions are evaluated by some MBB assembler routines invoked by the CDP.

Internally, the CDP represents the four memory spaces as part of one huge space. Microcode starts at 30000000, main memory at 50000000, dispatch at 60000000, and ALU scratch registers at 70000000. Thus, location 5 in main memory is represented as 30000005. If a symbol is used as a label, the high order bits of the symbol's value are set according to the address space. Since the CDP can thus associate address spaces with symbols, the user can use an abbreviated format for examining locations defined by labels. For example, to examine the microcode at address label "start", one may type "start/"; this command both opens the proper microcode address and sets the current address space to microcode memory.

The symbol "." has a value equal to the "current location". The "current location" normally represents the last location examined; however, after a ";C" command, the current location's high order bits are changed to reflect the new address space.

Values of locations in microcode memory are typed out, and read in, as microcode source statements. For example, changing the statement at label "FOO" from "0->L3" to "1->L3" might involve the typescript:

```
FOO/     0->L3     1->L3<return>
```

To enter an explicit constant, one can use the assembler pseudo-op EXP (Section 11.4):

```
FOO/     1->L3     EXP 3<return>
```

Various commands manipulate symbols. "=" asks the CDP to express numerically the last value typed in or out; "$=" asks the CDP to express the last value typed in or out with a symbol if possible. Here are some uses of "=":

```
20001 U/     40->BASE     =  36201400500
20000 U/     R0&MAR       =  0
START =      30020455
```

A microcode word of all zeroes is interpreted as the instruction "R0&MAR". "START" is a label at microcode address 20455; the high order 3 identifies the microcode memory space.

If "s" is a symbol, "s$K" half-kills "s" (suppresses its use in type-out) and "s:" defines s as a label at the current location.

The CDP assigns special meaning to the symbols designating the MBB's working registers (MAR, MBR, etc.). UDDT has no command to examine these registers; however, their values are saved in a block of ALU scratch registers when an exceptional condition occurs. Typing "MAR/" will examine the associated ALU scratch register; similarly with "MBR/" and so on. Thus, a snapshot of the micromachine's state at the last failure may be examined easily.

10.2.3 Controlling MBB Program Execution

We now consider the commands to control the MBB's program execution. Let e be an expression, and let n be the octal representation of its value. "e$G" starts the application software. When the user types "e$G", the CDP sends the MBB "nG". The CDP then sends the MBB "E" to make UDDT release control of the terminal interface to the application software. The CDP then enters application mode, in which it passes characters transparently between the user and the MBB; the user can then talk with the application software.

Typing control-X halts the application software. The CDP sends the MBB a control-N to regain the attention of UDDT, sends "H" to halt the application software, and reenters command level. (The CDP might already be in command level when control-X is typed, since the user may have just typed control-N to return the

CDP to command level. In that case, the CDP sends just "H" to the MBB.)

Typing "$P" proceeds after control-X or control-N. If the application software had been halted, the CDP wakes it up by sending "G", sends "E", and reenters application mode. If the user had not halted the application software, but merely returned with control-N to command level, the CDP simply reenters application mode.

Command level has some editing characters: control-A or control-H for character delete, and rubout for word delete.

```
";I" inputs a prepared file of commands.
";H" halts the CDP (returns to the monitor).
```

The CDP's miscellaneous error message is "XXX".

10.3 The Simulated MBB

Even while the CDP interacts with the real MBB, it keeps a simulated MBB in the background. Files loaded into the real MBB are also loaded into the simulator. If the user asks to examine the lower 8K of microcode memory, the CDP fetches the value from the simulator since this area cannot be read in the real MBB. When the contents of any memory are changed, the new value is entered into the simulator as well as the real MBB.

The ";O" command brings the simulator out of the background. The CDP enters a "simulator only" state, in which it communicates with the simulator only and it activates the simulator to execute microcode on command. A second ";O" deactivates the simulator and restores communication to the real MBB. More generally, each ";O" toggles the "simulation" state between using the real MBB and using the simulator. (The CDP can also be started up to always use the simulator only, by initially telling the CDP not to open a connection to the real MBB. Section 10.6 gives examples.)

The simulator provides debugging facilities not available on the real MBB, such as breakpoints, single-step execution, tracing, and the ability to examine the machine's complete internal state. On the other hand, the simulator has some disadvantages: it is slow, it does no I/O except for one of the terminal interfaces, and subtle timing considerations are different.

The CDP talks with the simulator much the way it talks with the real MBB. All of the modes and commands available with the real MBB are also available with the simulator. Some of these commands are implemented in the same way, by "sending" the same DDT commands to the simulator. For example, "e$G" still causes the CDP to send "nG" followed by "E" where n is the octal representation of the value of e. (If the simulator is not already running, the CDP first starts the simulator at microcode address 0, yielding a "Jumped to zero" message.) Other commands, from necessity or expediency, are implemented with short cuts by poking directly into the simulator's memory. For example, ";L" loads the simulator's memory directly from the file specified.

As one would expect, ";D" now lets the user talk directly with the simulator's DDT.

Certain commands have slightly different meanings. Control-X now halts the simulator completely, and "$P" resumes simulator execution after control-X or a breakpoint. Since the MBB's working registers (such as MAR) can now be examined directly, the CDP no longer looks in an ALU register block for the saved values. Control-R does not work with the simulator.

Breakpoints in the simulated MBB work as follows:

The command "y$B" sets a breakpoint at location y. Before the instruction at y is executed, the CDP types "BPT@y", halts execution, and returns to command level. If the command format "x<y$B" is used, the value of location x is also typed out at the breakpoint. If two escapes are used ("y$$B" or "x<y$$B"), the breakpoint is automatically "proceeded" after the typeout occurs. (The CDP neither halts execution nor returns to command level.) To delete the breakpoint at y, type "y$D". Any of these commands without arguments ("$B", "$$B", or "$D") delete all breakpoints.

After a breakpoint has occurred, "$X" executes the next microinstruction and "n$X" executes the next n microinstructions. The instructions executed are printed out, and the CDP stays at command level. "$P" resumes continuous execution and returns to application level.

The simulator permits examining and modifying the state of the microinterrupt system. Each microinterrupt level is known to the simulator by a number in the range 1-63. Higher numbers correspond to higher priorities. Those defined so far:

0	MAIN
1	software-controlled interrupt
61	terminal output
62	terminal input
63	100 microsecond clock interrupt

Level 0 (MAIN) is always set. The location MAXINT holds the number of the highest interrupt request set. "$I" clears all interrupts. "n$IS" and "n$IC" set and clear level n respectively.

There are two trace flags, both initially off. The first traces every instruction; the second traces only those instructions with labels. The flags are toggled by ";T" and ";;T" respectively.

The simulator detects various violations of microprogramming rules, such as disturbing the MAR or MBR during a main memory operation or failing to load the MBR before a main memory store.

The real MBB in such cases would not complain but would act unpredictably.) Such errors halt execution and return to command level. ";?" queries the CDP for a description of the last error.

"n$M" sets the data word width. n must equal 16 or 20. If a number, n is interpreted in decimal. Initially 20 is used. This command chooses 16-bit or 20-bit populated data paths for the simulated machine; the command is independent of the simulator's using the data width mode flags in MISC. (Section 7.3 explains the data width choices.)

";R" resets all I/O devices.

Table 10.1 shows special locations for examining the simulated machine's internal state:

ALURES	ALU output
ALUST	
BASE	
BASEDEL	the delayed version of BASE
BUTTON	set to push the button (must be explicitly cleared)
IOBUS	
IOSTE	IO state counter
LITES	
MAR	
MAROP	memory operation
MARTYPE	
MAXINT	current highest priority interrupt pending
MBR	
MIR	
MISC	
MISC2	
MSTATE	state counter of memory system
PADD	physical memory address
RAMADDR	
REGRES	data read from registers
REGSEL	absolute register number selected
SRCRES	contents of source bus
SYNERR	error syndrome bits
SYNSRC	syndrome source
TEMP	
TIME	increments every microcycle (never cleared)
UADD	address of microinstruction in UIR
UIR	microinstruction being executed
UPC	

Table 10.1: The Simulated MBB's Internal Registers

Various illogical attempted uses of the simulator will yield the CDP's general error message "XXX". One example is typing ";D", trying to talk with the simulator's DDT, if the simulator is not running.

10.4 Summary of Commands

Below is a list of the CDP's commands. Starred commands are for the simulator only. n indicates an expression. Parentheses indicate optional parts of commands.

<rubout>	Aborts the current typein (echoes XXX).
/	Slash opens a location. Locations in lower 8K microcode memory are always opened in the simulator. Other locations are opened in the physical MBB if selected, but all changes are made in the simulator as well.
<cr>	Carriage return closes the currently open location. If <cr> is preceded by an expression, the expression's value is stored in the location being closed.
<lf>	Line feed closes the currently open location and moves to the next. As with <cr>, a new value may be entered into the location being closed.
^	Up-arrow closes the currently open location and moves to the previous one. As with <cr>, a new value may be entered into the location being closed.
=	Displays the last typed value as a number.
$=	Displays the last typed value as a symbol if possible.
sss	Defines the label sss to have the value of the current location.
* ;?	Types a description of the last microprogramming rule violation.
* ($)$B	(see description of breakpoints)
;C	Sets the current address space. (The user then types U, M, D, or R for microcode memory, main memory, Dispatch memory, or ALU scratch registers, respectively.)

* ($)$D	(see description of breakpoints)
;D	Enters "system" mode, in which the user talks directly with the MBB's DDT.
n D/	Examines dispatch memory location n.
E	Transfers control of the MBB's terminal interface from DDT to the application program. (This command is used only after ;D, when in system mode; the command is actually interpreted not by the CDP but by the MBB.)
^E	Control-E complements the MBB's full-duplex/half-duplex echoing mode. The MBB initially treats terminal as full-duplex. A control-E is useful to prevent double echoing if the physical terminal is really half-duplex. (This command is used only in system or application mode. The command is processed by the MBB.)
(n)$G	Starts the application software and enters application mode. The application software probably interprets n as a starting macrocode address. If n is not supplied, the CDP uses the value from the last "$G" command.
;H	Done, escape to monitor.
;I	Accepts input from a file as if it were typed to the CDP on the user's terminal.
* $I	Clears all interrupts
* n$IC	Clears interrupt level n.
* n$IS	Sets interrupt level n.
(n)$J	Jumps to microcode address n (0 if no address given).
s$K	"Half-kills" symbol s. Then s is available for typein but not typed out.
;L	Loads binary files. Symbols and simulated memory are always loaded; the physical MBB is loaded if selected.
* n$M	[n = 16 or 20] Sets data width to n.

n M/	Examines main memory location n.
^N	Control-N returns to command level. After control-N, use $P to return to application mode or control-X to halt.
;O	Selects simulator-only state, if running with a physical MBB. A second ;S will switch back to the physical MBB.
$P	Proceeds after breakpoints, control-X, or control-N.
* ;R	Resets all I/O devices
* n;R	Resets I/O device n.
n$R	Sets radix for typein and typeout to n. (If a number, n is interpreted in decimal.)
n R/	Examines ALU scratch register n.
;S	reads in the symbols from an MBB source file
* ;T	Toggles the full trace mode flag, to trace all instructions executed (initially off).
* ;;T	Toggles the symbol trace mode flag, to trace instructions whose locations have labels (initially off).
n U/	Examines location n in the microcode address space.
^X	Control-X is handled at interrupt level by the CDP. It puts the CDP at command level (stopping any loading), halts the real MBB's application software if running (when talking with the real MBB), and halts the simulator if running.
* $X	Single steps (execute a single microinstruction).
* n$X	Single steps n times.

10.5 Current CDP Bugs and Confusion

1) When two symbol files are loaded, suppose symbol "s" is redefined from value m to n. Symbol "s" still retains the ghost of its old value. Though "s=" will produce n, "m$=" will still produce "s".

2) When the CDP has unwittingly left the MBB in echo mode (perhaps after a "panic" input of five control-X's), it misinterprets echoes; for example, if "M" is echoed, the CDP will believe that the MBB is signaling a main memory error. Typing ";D" followed by "control-N" will correct this.

3) If the MBB's application software puts the terminal interface handler's routine "ttin.poll" in "transparent" posture, the CDP will get no response to its control-N when the user issues a control-N or control-X command. (The program listing explains "transparent" posture in discussing the terminal handler.) If the application software improperly keeps "ttin.poll" in transparent posture, push the MBB's button to recover.

4) Subtle timing considerations (such as the danger of the MBB missing input while waiting for output) work differently with the simulator than with the real MBB.

5) The simulator does not insert delay between the

microcode's setting or clearing the software-controlled interrupt and the value of INTS changing.

6) Although the following sequence will load paper tape into the H316 emulator:

```
;DDT (MBB)
!1G
E
^S^RPaper Tape Input file:
```

the following will not:

```
1$G
^RPaper Tape Input File;
```

Further, tape loading with control-R does not work with the simulator.

10.6 Invoking the CDP

To use the CDP and connect to an MBB, one must be a NETWIZ. Anyone can use the CDP in simulator-only mode. When you connect to the MBB, CPD asks for your name. If someone else then tries to connect to that MBB, they are informed by CDP that the machine is in use and who is using it.

Below are two typescripts of CDP use on system BBN-A. The first example uses the simulator only; the second uses the real MBB. File "<mbb>mbbimp.mbn" is a program file with microcode for both the system and the H316 IMP application. The file "ddt.mbn" is an H316 stand-alone DDT routine which starts at 20000. In both examples, the last line shows communication with the application software (with the macrocode). The "Jumped to zero" message in the first example comes from the CDP initially starting the simulator.

```
@<mbb>mbbsys.EXE.27
Open connection to MBB? (Y/N) n
MBB Symbolic DDT
;loader -- MBB Input File:  <mbb>mbbimp.MBN.1 [Old Generation]
Done.
;loader -- MBB Input File:  <mbb>ddT.MBN.1 [Old Generation]
Done.
20000$g
Jumped to zero
1/ 10057

@<mbb>mbbsys.EXE.27
Open connection to MBB? (Y/N) y
MLC# 3 Port# 71
Your name please:  phil in office (x1234)
Assigned to TTY161
MBB Symbolic DDT
;loader -- MBB Input File:  <mbb>mbbiMP.MBN.1 [Old Generation]
Translating file...
Loading boot routine...
Loading loader...
Starting load...Done.
;loader -- MBB Input File:  <mbb>ddT.MBN.1 [Old Generation]
Translating file...
Loading boot routine...
Loading loader...
Starting load...Done.
20000$g
1/ 10057
```

11 THE MBB ASSEMBLER

MBB microcode and data are assembled by a two-pass assembler which runs on TENEX and TOPS-20. The assembler can be called by "<mbb>mbbasm.exe". A single source file is accepted as input by the assembler, and a binary file, an error file, and a listing file may be produced.

When processing instructions to go in the microcode address space, the assembler expects to see a microcode instruction; when processing an instruction to be in any other address space, the assembler expects to see an arithmetic expression. These expectations can be changed somewhat, as discussed later, by parentheses, square brackets, and the EXP pseudo-op.

After some preliminary matters, we will discuss in detail microcode instructions and pseudo-ops.

11.1 Format Considerations

Letters are converted to upper case on input. Spaces and tabs are ignored. A semicolon starts a comment, causing the rest of the line to be ignored.

Symbols are distinguished by their first ten characters. In symbols, legal characters are letters, ".", "'", and "@". Labels are distinguished by their first six characters, and may not have more than ten characters.

If a source line has over 80 characters, the line is ignored and a warning is printed in the error file.

A source line is either a symbol definition (using "=" or "==", or using ":", or "::" to define labels), an instruction, or a pseudo-op. (Instructions may start with one or more labels.)

Symbols may be defined by "=" or "==":

```
A =  3
A == 3
```

With "==", the symbol is "half-killed". Half-killed symbols are not used in typeout by the CDP (Section 10.2.2). Similarly, a label defined with "::" instead of ":" is half-killed.

11.2 Numbers and Expressions

The RADIX pseudo-op sets the prevailing radix (the default is octal).

Numbers and symbols may be composed into expressions using the following operators:

```
= - * &(AND) !(IOR) ?(XOR) _(shift left) (NOT)
```

All are binary operators, except for the unary operator (bitwise complement) and - (minus or negative), which can be binary or unary. The unary operators are right associative, and are evaluated before binary operators. Thus -7 = (-7) = 6 and 7+1 = (-8)+1 = -7. Binary operators are left associative. Thus, 2*3+4 = 6+4 = 10 and 2+3*4 = 5*4 = 20.

Within an expression, square brackets enclose material to be evaluated as a microcode instruction.

11.3 Microcode Instructions

Generally, microcode instructions take the form:

```
[IF[NOT] <cond>] <reg> <op> <source> -> <reg>, <dest> [LS]
```

Most of the fields may be omitted under the proper circumstances. The register and source fields on the left side of the arrow may be interchanged, as may the register and destination fields on the right. Excess spaces and tabs are ignored. A label may precede an instruction, followed by 1 or 2 columns. A semicolon starts a comment; the rest of the line is ignored. Further details on the fields of instruction are given in the ensuing sections; the instruction syntax is defined formally in Section 11.5. Here are some examples:

```
          L0&7 -> L0
          7&L0 -> L0, MBR
          7-L0 -> L0
          R17+3 -> TEMP, R17
          R17+3 -> R17, TEMP
          IF ZERO G5 -> MAR(R)
          IFNOT ZERO G5 -> MAR(RIO)
          MAR(SL)
          DISP(3) -> UPC
          G0&100  LS
  LABEL:  1 -> G0
  LABEL:: 77000 -> G0
          20540 -> UPC
          G0& (1!2) -> G0
          NOT TEMP -> G0      ;COMMENT
```

Spaces and tabs are ignored. A semicolon starts a comment; the rest of the line is ignored.

11.3.1 Sources and Destinations

The ALU sources and operation appear to the left of the arrow; the places to write the ALU's output appear to the right. The two ALU inputs (the source and the scratch pad register), if both specified, may be given in either order. Similarly, if two places to write the ALU result are specified (a destination and a scratch register), they may be given in either order.

The source and destination mnemonics are given in Section 6.

11.3.2 Constants

Any permitted constant value is assembled properly as a source; illegal constants are flagged. Even though the code may be designed to run in 16-bit data width mode, constants are assembled for 20 bits. Thus, 177776 is an illegal constant; -2 however is legal, producing 3777776, which is functionally equivalent in 16-bit mode.

11.3.3 ALU Registers and Operations

The symbols L0 through L15 indicate "local" scratch register addresses 0 through 15 mapped by BASE; G0 through G15 indicate addresses 0 through 13 unmapped; MIRRA and MIRRB indicate address selection by an MIR daughterboard field mapped by BASE. (In general, Rn specifies value n for the microinstruction's five-bit Register field. If the source statement specifies no scratch register, the assembler uses a value of 0.)

The ALU operations are:

+	ADD
-	SUBTRACT (register minus source bus)
&	AND
!	INCLUSIVE OR
?	EXCLUSIVE OR
NOT	COMPLEMENT source bus
	PASS source bus

If the scratch register input to the ALU (The ALU's "A" input) is missing in the source statement, and the NOT operator is not present (e.g., "MBR->TEMP"), a PASS operation is assembled. If the source bus input is missing (e.g., "R0->MAR"), the assembler assembles an ADD operation with a constant of 0 to drive the source bus. If the source statement contains no arrow (e.g., S16MBR L5), no destination for the ALU result is specified; the MBB will still perform the indicated operation, latching the ALU status if desired.

Within a microcode instruction, operators are normally interpreted as ALU operations; inside parentheses, however, operators are interpreted as forming arithmetic expressions. Thus, "(1+2)->MAR" passes the constant 3 to the MAR but "1+2->MAR" is illegal since it specifies an ALU operation between two constants (instead of, for example, between a constant and a scratch register).

11.3.4 Conditional Execution

A conditional execution is indicated by preceding the body of the statement with IF or IFNOT followed by one of these conditions:

TRUE	(always true)
ZERO	ALU status register has the "zero" bit on
NEG	ALU status register has the "negative" bit on
ODD	ALU status register has the "odd" bit on
MARCOND	MAR condition (from the MARDB)
INTP	an interrupt other than the software-controlled interrupt has been requested
MODEF0	mode flag 0 (in MISC) is on

If no condition is specified, "IF TRUE" is assembled.

11.3.5 Other Special Cases

Certain sources and destinations take qualification codes enclosed in parentheses. When referenced as a destination, the MAR optionally takes codes:

R	read "macro"
RIO	read I/O
RP	read "physical"
W	write "macro"
WIO	write I/O
WP	write "physical"

(These symbolic codes have values equal to the value to put in the microinstruction's Transfer field.) When referenced as a source, the MAR optionally takes codes:

SR	shift right
SL	shift left

(These symbolic codes have values equal to the value to put in the microinstruction's Constant field.)

When referenced as a source, Dispatch Memory takes an 8-bit qualifying code (between 0 and 255) to be interpreted by the application-dependent MIR daughterboard. (This code specifies the value for the microinstruction's Constant field.)

A "LS" at the statement's end indicates latching the ALU status.

Further information on source statement format, such as restrictions on the constants and the dispatch source, can be found in Section 2.7.

11.4 Pseudo-Ops and Address Spaces

Below are the pseudo-ops, with their formats and use. The "pointer" referred to below is the next location to be assembled; it is represented in the assembler by the symbol ".".

EXP e	Assemble the value of expression e in the next location. (Do not interpret e as an instruction.)
DISPATCH n	Set the pointer to Dispatch address n.
END	Ignore all succeeding input.
INCLUDE "F"	Textually insert source file F after the line with the include.
[NO]LIST	Disable or enable the listing output.

MAINM n	Set the pointer to main memory address n.
REGS n	Set the pointer to ALU register address n.
UPROM n	Start assembling at microcode address n.
URAM n	Set the pointer to microcode address n!20000. (20000 is the start of RAM.)

When assembling for the microcode address space, the assembler interprets source lines as microcode instructions. Thus,

```
0
```

is interpreted as

```
0 ->
```

To assemble the explicit constant "0", write

```
EXP 0
```

(The instruction "0 ->" functions as a no-op. In fact, the assembler symbol "NOP" is defined as 0. Thus, in the microcode address space, the source statements "0" and "NOP" both assemble as no-ops. (The explicit constant 1 ("EXP 1") also functions as a no-op; "EXP 0" has the wrong parity.)

In other address spaces, the assembler interprets source lines as expressions rather than microcode instructions. To have text interpreted as a microcode instruction, enclose it in square brackets. Square brackets could be used to store a microinstruction in main memory. Thus,

```
MAINM   0
([3->MAR]_-20) & 177777
[3->MAR] & 177777
```

would store the microcode instruction "3->MAR" in main memory locations 0 and 1 (16 bits in each.)

11.5 The Parser

Table 11.1 gives the assembler's parser.

```
 L <- L1 EOL

 L1<- <sym> = E
   <- <label>: L2
   <- L2

 L2<- <keyword> <args>
   <- I  ;legal only in microcode address space
   <- E  ;not legal in microcode address space
   <- --

 I <- IFP SRC -> DEST FLG
   <- IFP SRC FLG

IFP<- IF <cond>
   <- IFNOT <cond>
   <- --

SRC<- S ! R ! S <ALU op> R ! R <op> S ! NOT S

 R <- <reg. spec.>

 S <- <source spec> !  <source spec> (E) ! T

DEST<- D ! R ! D,R ! R,D ! --

 D <- <dest. spec.> !  <dest spec>(E)

FLG<- -- ! <flag spec> FLG

 E <- T
   <- T { <binary op> T }...

 T <- <unary op> T
   <- (E)
   <- [I]
   <- <symbol>
   <- N
   <- .

   <- <number>
```

Table 11.1 Assembler Grammer

11.6 Invoking the Assembler; Files Generated; Loading

Here is a sample typescript on system D:

```
@<mbb>mbbasm.EXE.5
Source file:  foo.MIC.1 [Old Generation]
Binary file:  FOO.MBN.1 [New file]
Error file:  TTY:FOO [confirm]
Listing file:  FOO.LST.1 [New file]
Pass 1
Pass 2
@
```

As shown, the error file indicates the start of the two passes. Any errors are noted in the file as they are discovered. Some errors are noted on both passes, and some only on Pass 2.

The default names for the binary and listing files are used by typing escape after the prompt. Typing an escape for the error file causes the output to default to the user terminal. To send the errors to a disk file, the filename must be proceeded by 'DSK:'.

The listing file contains all source lines shifted right. If a line contains an instruction, the octal address is given the left, with high order bits indicating the memory space according to this code:

300000000	microcode memory
500000000	main memory
600000000	Dispatch memory
700000000	ALU scratch pad registers

The location contents are not given in the listing file.

The binary file contains the assembled addresses and values, with a symbol table at the end for use by the CDP.

The CDP can load a binary file into the MBB. All address spaces may be loaded, with a few caveats. First, the lower 8K of microcode cannot be loaded, since the MBB has no loading path for it. The CDP does not try to load these addresses into the real MBB, though it loads them into the simulated MBB. Second, if the binary file specifies loading any ALU scratch registers used by the MBB system code, problems will occur. Third, as mentioned in Chapter 8, the loadable system microcode must be loaded before Dispatch memory, can be accessed.