

Introduction to Microcoded Implementation of a CPU Architecture

N.S. Matlo

January 21, 1997

1 Hardwired Versus Microcoded Implementation

Strictly speaking, the term *architecture* for a CPU refers only to “what the assembly language programmer” sees—the instruction set, addressing modes, and register set. For a given *target* architecture, i.e. the architecture we wish to build, various implementations are possible. We could have many different internal designs of the CPU chip, all of which produced the same effect, namely the same instruction set, addressing modes, etc. The different internal designs could then all be produced for the different models of that CPU, as in the familiar Intel case. The different models would have different speed capabilities, and probably different prices to the consumer. But the same machine language program, say a .EXE file in the Intel/DOS case, would run on any CPU in the family.

There are two major approaches to developing these internal designs, one being *hardwired* and the other the *microcoded* approach. The former is the more “straightforward” one, in the sense that it is a straight digital design (and layout) problem. It is feasible for simple CPUs, as is the case for today’s RISC (Reduced Instruction Set Computer) processors. However, for CISC (Complex Instruction Set Computer, such as the Intel family) processors, hardwired design takes so long that a “shortcut” is often used, in the form of microcoded design.

In the latter, we first build a very simple CPU, called the *microengine* (itself implemented via the hardwired approach), and then write a program to run on it. The function of that program, called the *microcode*, is to implement the target architecture. In essence, the microcode is a simulator for the target architecture. (Or one can view the microcode as similar to the interpreter code for an interpreted language such as BASIC or Java.)

Different designs within the same CPU family would have different microengines. The microengines could differ in many respects, such as in the number of internal buses (the more the better, since more buses means that more transfers among registers can be done in parallel). Of course, having different microengines means having different microcode even though the target architecture is the same.

Another consideration is the size of the microcode. The microcode is stored in a section of the CPU called the *control store* (since it is controlling the operation of the target architecture). The faster models in the family may need larger control stores, which produces space problems of various kinds. In a multichip CPU, for instance, large control stores increase the chip count (and thus the manufacturing costs), while for a single-chip CPU, we may have trouble fitting the control store into the small area (“real estate”) a chip provides.

The execution of an instruction in the target architecture occurs via the execution of several instruc-

tions (*microinstructions* in the microengine. For example, suppose again that our target family is Intel. Suppose the user is currently running a game program. What actually occurs inside?

The game program is composed of Intel machine language (produced either by the compiler or assembler from the game programmer's source code). This program is stored in the RAM of the computer, having been loaded there by the operating system when the user gave a command to play the game. One of the game program's instructions might be, say, MOV AX,3, which copies the value 3 to the AX register. The microcode inside the CPU would make the CPU fetch this instruction from RAM and then execute it; altogether there might be seven or eight microinstructions executed in order to fetch and execute MOV AX,3. Note that the microengine fetches the microinstructions from the control store, and causes their execution.

As mentioned earlier, the microcode "shortcut" shortens the development time for a CPU, which may be an important consideration, e.g. when market share is at stake. However, a microcoded CPU is typically slower-running than a hardwired one. The control store is usually implemented as a ROM, microinstruction fetches from which slow down the show.

Some of the VAX-family machines implemented the control store in RAM, loaded at boot time, so as to enable the user to add instructions to the VAX instruction set by adding to the microcode. If for example a user anticipated doing a lot of sorting in his/her software, the user could add a QSORT instruction to the VAX instruction set, performing Quicksort. The basic Quicksort algorithm would still be used, but by implementing it in microcode instead of in VAX machine code, performance should be faster. There would be many fewer instruction fetches from the system (i.e. external to the CPU) RAM, and in addition microcode allows multiple simultaneous register transfers, something which of course could not be done at the machine language level. Of course, assemblers would not recognize the new instruction, nor would compilers generate it, but one could make a separate subroutine out of it and put calls to the routine in one's assembly or high-level language source code. But users found that it was not worth the trouble.

By the way, the microcode has an interesting "Is it fish or fowl?" status. On the one hand, because it is a program, it might be considered software. Yet on the other hand, because it is used to implement a CPU (the target CPU) and is typically stored in a ROM, it might be considered hardware. This duality has created much confusion in court cases involving work-alike chips.

2 Example: MIC-1 and MAC-1

The machines in the example here come from the text *Structured Computer Organization*, by Andrew Tanenbaum (Prentice-Hall, 1990). This has been one of the most popular undergraduate computer architecture texts in the nation, so much so that MIC-1 and MAC-1 became important teaching tools in their own right, independent of the textbook. A number of simulators have been written for these machines.

MAC-1 is the target architecture, and MIC-1 is the microengine.

2.1 MIC-1

2.1.1 Hardware

MIC-1 is Tanenbaum's example microengine. It is loosely modeled after the AMD 2900 bitslice microprocessor series.¹

The MIC-1 operates on 16-bit words, has 16 registers and 32-bit instructions. A block diagram appears at the end of this document.

In keeping with its eventual "CPU within a CPU" status, MIC-1 is a CPU microcosm. Just like other CPUs, it has a program counter, labeled "mpc" for "microprogram counter." The program which runs on it, the microprogram, is of course stored in memory. That memory is typically called the *control store*, since it controls the operation of the outer (i.e. target) CPU. Just as the program counter in other machines serves as a pointer to the current instruction in memory, the microprogram counter here points to the current microinstruction in the control store. Just as other machines have instruction registers (IR) in which to hold the currently-executing instruction, here we have a microinstruction register (mir) to do this for the current microinstruction. By the way, the mir here has been drawn to display the various bit fields (amux, cond, etc.) within the MIC-1 instruction format.

The incrementer, also seen in the diagram, adds 1 to the mpc to go sequentially through the microinstructions in the control store, just as program counters in other CPU's are incremented after each instruction. And, also as in other machines, we occasionally encounter a branch. At the microengine level, branches are typically "folded" in with other microinstructions; each microinstruction includes a branch, and a condition field indicating whether or not the branch will actually be taken. In MIC-1, the corresponding fields are cond and addr. The digital logic (consisting of a multiplexor and miscellaneous logic) in mmux chooses between the incremented mpc value or the branch address in addr.

Again, our eventual goal is to use MIC-1 to build our target machine, which will be named MAC-1. It is important to note that these two machines will share some components in common. For example, the register file we see in MIC-1 (which consists of 16 16-bit registers) will serve as the registers not only for MIC-1, but also for MAC-1 as well.

The same holds true for the ALU (labeled "alu" in the diagram), and for the n and z bits which are produced as a byproduct of every ALU operation. The n bit will be 1 if the ALU operation produces a negative number (even if the bit string is not intended as a number, say in the case of a fetched MAC-1 instruction), and will be 0 otherwise. Similarly, the z bit will be 1 if and only if the ALU produces the bit string which is all 0s. We will assume here that n and z are recorded in flip-flops.

The entire diagram would be on a single chip (assuming a single-chip CPU, as is almost always the case these days). The contents of the control store determine what architecture this chip implements. If the control store contains the microcode for MAC-1, then this chip will be a MAC-1 chip.

Note that in the diagram, mar (memory address register) and mbr (memory buffer register, i.e. memory data register)² will be connected to the address and data pins of the chip, respectively. Those pins are connected to MAC-1's system bus' address and data lines, and since MAC-1's

¹Not to be confused with the later AMD 29000, a RISC machine which is quite different from the 2900.

²Review these terms carefully from your ECS 50 or EEC 70 course.

memory is also connected to these lines, mar and mbr serve as the MAC-1's chips gateways to memory.

MIC-1 features 16 registers (marked "register file" in our diagram). Unfortunately, instead of naming them, say, r0, r1, ..., r15, Tanenbaum gave the registers names chosen according to their anticipated usage in MAC-1. For example, register 0 is used as the program counter in MAC-1, so Tanenbaum named it "pc." This is misleading, because we ought to be able to use MIC-1 to build other target architectures besides MAC-1, and some of them may have different register names and functions than MAC-1's. We are stuck with Tanenbaum's names, but keep in mind that we can use those registers for other things, in spite of their names.

| reg number | Tanenbaum name | comments |
|------------|----------------|-------------------------|
| ----- | ----- | ----- |
| 0 | pc | MAC-1's program counter |
| 1 | ac | MAC-1's accumulator |
| 2 | sp | MAC-1's stack pointer |
| 3 | ir | |
| 4 | tir | |
| 5 | 0 | hardwired value 0x0000 |
| 6 | +1 | hardwired value 0x0001 |
| 7 | -1 | hardwired value 0xffff |
| 8 | amask | hardwired value 0x0fff |
| 9 | smask | hardwired value 0x00ff |
| 10 | a | |
| 11 | b | |
| 12 | c | |
| 13 | d | |
| 14 | e | |
| 15 | f | |

Similarly, since MAC-1 has 12-bit addresses, Tanenbaum set MIC-1's pc, sp and mar registers to be 12 bits wide.

There are three buses in the diagram, labeled 'a,' 'b' and 'c.' The first two of these form input to the ALU from the registers, while the c bus is for output from the ALU back to one of the registers. Again, keep in mind that these are internal buses, i.e. buses inside the MAC-1 chip. MAC-1 will then have an external system bus which connects the MAC-1 chip with MAC-1 memory and I/O ports.

It should be noted that these internal buses would probably exist even if the MAC-1 chip implementation were hardwired. We do need some way to transfer data internally between registers, and if we did not use buses, we would need extra multiplexers, which would occupy too much chip space.

By the way, our diagram is missing a connection from the c bus to the register file; please draw it in.

MIC-1 features a four-phase clock. A regular clock, i.e. a crystal oscillator, is fed into the box labeled "subcycle gen." The latter consists of circuitry which outputs subcycle signal lines 1, 2, 3 and 4, which are asserted in cyclical fashion, i.e. 1, 2, 3, 4, 1, 2, 3, 4, ... The subcycle generation box can be constructed, for instance, using *delay elements*: The input line, from the crystal oscillator,

is split into four lines. The first of these lines leads directly to the subcycle 1 output line, but the second leads into a delay element whose output is connected to the subcycle 2 output line. The third line leads into two delay elements connected in series, the output of which becomes the subcycle 3 output line, and similarly three delay elements are used for subcycle 4. The delay elements give us the sequencing we need among the four subcycles.

The program which runs on a microengine is called the *microprogram* or *microcode*. It is stored in the *control store*, typically implemented as a ROM. The micro-PC mpc contains the address in the control store of the current microinstruction, which is fetched from the control store and places in a temporary holding register, mir.

Here are the roles of the four subcycles:

1. The current microinstruction is fetched from the control store into mir.
2. The registers (if any) which are specified in the microinstruction are copied to the a and b buses, and captured in the two bus latches.
3. The ALU uses this time to produce a stable output from its two inputs.
4. Store ALU output to mbr and register, if specified.

Instead of the usual format of op code and so on, microinstructions typically have fields to control components directly. For example, in mir we see fields labeled amux, alu, etc. whose bits are connected to wires which do control these components. Here is a summary of the fields in the instruction format:

| | |
|---|---|
| amux (bit 31) ----- 0 = input from 'a' latch 1 = input from mbr | cond (bits 30-29) ----- 00 = no branch 01 = branch if n = 1 10 = branch if z = 1 11 = unconditional branch |
| alu (bits 28-27) ----- 00 = a + b 01 = a and b 10 = a 11 = not a | sh (bits 26-25) ----- 00 = no shift 01 = shift right 1 bit 10 = shift left 1 bit 11 = (not used) |
| mbr (bit 24) ----- 1 = load mbr from shifter 0 = do not load mbr | mar (bit 23) ----- 1 = load mar from B latch 0 = do not load mar |
| rd,wr,enc (bits 24-20) ----- 1 = yes, 0 = no | c,b,a (bits 19-8) ----- 4-bit register number |

```

addr (bits 7-0)
-----
branch target

```

The fields a, b and c in the instruction each consist of four bits, to indicate one of the 16 registers. The ‘a’ field indicates which register will be copied to the ‘a’ bus, and the b field does the same thing for the b bus. The c field controls which register the ALU output is copied to. Accordingly, the a, b and c fields in mir lead to the three decoders at the top of the diagram, whose outputs in turn lead to tri-state devices at the register file, so that the proper registers are accessed.

The c field could have been 16 bits wide, allowing multiple registers to be loaded simultaneously from the c bus. This would be useful, for instance, if we would like to have a target-architecture instruction that does something like the two Intel instructions MOV AX,CX and MOV BX,CX, which load the AX and BX registers from the CX register, in a single target-architecture instruction. This would give us a faster machine, at the expense of having wider microinstructions and thus a more space-hogging control store. By the way, this wider instruction format is said to be more *horizontal*, whereas the original one is called more *vertical*.³

Note that the c decoder appears to be, and is, more than just an ordinary decoder, since it has additional input lines—enc and the subcycle 4 clock line. In some instructions we will not want the ALU output to go to any of the registers; we may just want it to go to the mbr. The enc field allows us to control this.

The purpose of the subcycle 4 line input to the c decoder is more subtle. Suppose, for example, that our instruction will add registers 3 and 8, and then store the sum back into register 3. Suppose further that this will be done by letting register 3 flow out onto the ‘a’ bus (i.e. we would put 0011 in the ‘a’ field in the microinstruction) and copying register 8 to the b bus (1000 in the b field). During subcycle 1, the output of the ALU will be garbage, and if we are not careful, then that garbage will be then stored into register 3. Or, early in subcycle 1 the c field in mir will be garbage, and this might lead to some register other than register 3 being falsely written to. The subcycle 4 line input to the c decoder prevents disasters such as these.

Note too that the values on the ‘a’ and b buses are latched, so that we avoid “feedback loop” problems. Say for instance that in the above example registers 3 and 8 contain 25 and 54. The sum, 79, will be put into register 3, as desired, but if we are not careful, that new value for register 3 will flow out onto the ‘a’ bus, and the alu will computer $79+54 = 133$ and put 133 into register 3, etc. The latches prevent this, as would use of edge-triggered or master-slave flip flops in the registers.

You should scrutinize the diagram for other elements of the design which were motivated by timing considerations. For instance, note that the subcycle 1 input which allows the loading of mir is needed. Without it, the mir would be continually loaded from the control store, and as mpc changes in later subcycles, mir would then change too!

Tri-state devices will be used in many places in the overall circuit. In the mir example in the preceding paragraph, for instance, each of the 32 bits of mir would be guarded by a tri-state device, the latter being controlled by the subcycle 1 input.

The subcycle time would have to be set to the duration of the longest operation. For example, since the ALU and shifter must produce valid results by the end of subcycle 3 (to provide valid

³The Intel Pentium, for instance, is very horizontal, with 118 bits per microinstruction.

inputs to subcycle 4), and since the ALU's inputs are not valid until the end of subcycle 2, that means that both the ALU and the shifter must do their work within subcycle 3. So, the subcycle time must be at least as long as the sum of the maximum ALU time and the shift time. If some other operation takes longer, e.g. the mir load from the control store, then the subcycle time would need to be even longer.

2.2 Microprogramming

Programming in machine language, either with 0's and 1's or in hex, would be extremely tedious. That is why assemblers were invented, so as to replace the numbers with something that looks like English. For the same reason, microprogrammers use a microassembly language. Tanenbaum calls the microassembly language he invented for MIC-1 "mal."

In order to introduce mal, here is an example instruction:

```
mar:=sp; mbr:=ac; wr; goto 10;
```

(Note that the 10 here means location 10 in the control store.)

What will this assemble to? Well, the field `mar:=sp` says that we wish to copy `sp` (register 2) to the `mar`. As you can see, the only bus connection to `mar` is that of the `b` bus. Thus the assembler will put 0010 in the `b` field of the instruction. It will also put a 1 in the `mar` field. Similarly, for `mbr:=ac` it will put 0001 in the 'a' field (since `ac` is register 1), 1 in the `mbr` field, and 10 in the `alu` field. For `wr`, it will put 1 in the `wr` field, and 0 in the `rd` field. For `goto 10`, it will put 11 in the `cond` field and 00001010 (the binary coding for the decimal number 10) in `addr`. Since nothing is to be written back to the register file, `enc` will be set to 0; this also allows us to put anything at all in the `c` field, but we might as well make it 0000. No shifting is to be done, so we put 00 in the `sh` field. The coding for the entire instruction will then be 71a0210a.⁴

You can learn the other mal mnemonics by looking at the microcode for MAC-1, presented in the next section. Note that "band" refers to "boolean and," the logical AND-ing operation, and "inv" is inversion, i.e. logical complement. The construct "`alu:=`" means that an ALU operation will be performed on the specified operands, but that neither the `mbr` nor a register will be loaded.

2.3 MAC-1

2.3.1 Architecture

MAC-1's register set consists of a program counter PC, a stack pointer SP, and an accumulator AC. The latter is intended as a general-purpose register. Addressing modes are direct, indirect and local.

Local addressing, which is common in many architectures, is a stack-relative mode intended for accessing local variables. Suppose for example we have a C program with a function named, say, `wxyz()`, with one local-variable declaration,

```
int r,s;
```

⁴Make sure you verify this for yourself. Note too that other codings would be equally valid, depending for instance on what value we put into the "don't care" field for `c`.

Then the compiler, when compiling `wxyz()`, will put in an instruction

DESP 2

which will leave two words on the stack for `r` and `s`, with `s` at `sp+0` and `r` at `sp+1`. Then to add, say, `r` to the accumulator, the compiler will generate code

ADDL 1

MAC-1's memory consists of 4096 16-bit words, and the `mar` register is accordingly 12 bits wide. (Thus `mar`'s connection to the `b` bus is only to the less-significant 12 lines of that bus.) Memory access is assumed to take two full cycles within functions. Memory-mapped I/O is used.

The instruction set is summarized below; note that all instructions are 16 bits in length.

| machine language | mnemonic | long name | action |
|------------------|----------|-----------------|----------------------------|
| ----- | ----- | ----- | ----- |
| 0000xxxxxxxxxxxx | LODD | load direct | ac:=m[x] |
| 0001xxxxxxxxxxxx | STOD | store direct | m[x]:=ac |
| 0010xxxxxxxxxxxx | ADDD | add direct | ac:=ac+m[x] |
| 0011xxxxxxxxxxxx | SUBD | subtract direct | ac:=ac-m[x] |
| 0100xxxxxxxxxxxx | JPOS | jump positive | if ac >= 0 then pc:=x |
| 0101xxxxxxxxxxxx | JZER | jump zero | if ac = 0 then pc:=x |
| 0110xxxxxxxxxxxx | JUMP | jump | pc:=x |
| 0111xxxxxxxxxxxx | LOCO | load constant | ac:=x, x unsigned |
| 1000xxxxxxxxxxxx | LODL | load local | ac:=m[sp+x] |
| 1001xxxxxxxxxxxx | STOL | store local | m[x+sp]:=ac |
| 1010xxxxxxxxxxxx | ADDL | add local | ac:=ac+m[sp+x] |
| 1011xxxxxxxxxxxx | SUBL | subtract local | ac:=ac-m[sp+x] |
| 1100xxxxxxxxxxxx | JNEG | jump negative | if ac < 0 then pc:=x |
| 1101xxxxxxxxxxxx | JNZE | jump nonzero | if ac != 0 then pc:=x |
| 1110xxxxxxxxxxxx | CALL | call procedure | sp:=sp-1; m[sp]:=pc; pc:=x |
| 1111000000000000 | PUSHI | push indirect | sp:=sp-1; m[sp]:=m[ac] |
| 1111001000000000 | POPI | pop indirect | m[ac]:=m[sp]; sp:=sp+1 |
| 1111010000000000 | PUSH | push | sp:=sp-1; m[sp]:=ac |
| 1111011000000000 | POP | pop | ac:=m[sp]; sp:=sp+1 |
| 1111100000000000 | RETN | return | pc:=m[sp]; sp:=sp+1 |
| 1111101000000000 | SWAP | swap ac and sp | tmp:=ac; ac:=sp; sp:=tmp |
| 11111100yyyyyyyy | INSP | increment sp | sp:sp+y, y unsigned |
| 11111110yyyyyyyy | DESP | decrement sp | sp:sp-y, y unsigned |

2.3.2 MIC-1 Implementation of MAC-1

Here is the microcode for the implementation:

```

0 BEGIN:      mar:=pc; rd;
1             pc:=pc + 1; rd;
2 L0or1:      ir:=mbr; if n then goto L10or11;
3 L00or01:    tir:=lshift(ir + ir); if n then goto L010or011;
4 L000or001:  tir:=lshift(tir); if n then goto L0010or0011;

```

```

5  L0000or0001:  alu:=tir; if n then goto STOD;
6  LODD:        mar:=ir; rd;                                #0000 = LODD
7  contLODL:    rd;
8              ac:=mbr; goto BEGIN;
9  STOD:        mar:=ir; mbr:=ac; wr;                        #0001 = STOD
10 contWR:      wr; goto BEGIN;
11 L0010or0011: alu:=tir; if n then goto SUBD;              #ADDD or SUBD?
12 ADDD:        mar:=ir; rd;                                #0010 = ADDD
13 contADDL:    rd;
14              ac:=mbr + ac; goto BEGIN;
15 SUBD:        mar:=ir; rd;                                #0011 = SUBD
16 contSUBL:    ac:=ac + 1; rd;
17              a:=inv(mbr);
18              ac:=ac + a; goto BEGIN;
19 L010or011:   tir:=lshift(tir); if n then goto L0110or0111;
20 L0100or0101: alu:=tir; if n then goto JZER;
21 JPOS:        alu:=ac; if n then goto BEGIN;              #0100 = JPOS
22 contJUMPS:   pc:=band(ir,amask); goto BEGIN;
23 JZER:        alu:=ac; if z then goto contJUMPS;          #0101 = JZER
24              goto BEGIN;
25 L0110or0111: alu:=tir; if n then goto LOCO;
26 JUMP:        pc:=band(ir,amask); goto BEGIN;            #0110 = JUMP
27 LOCO:        ac:=band(ir,amask); goto BEGIN;            #0111 = LOCO
28 L10or11:     tir:=lshift(ir + ir); if n then goto L110or111;
29 L100or101:   tir:=lshift(tir); if n then goto L1010or1011;
30 L1000or1001: alu:=tir; if n then goto STOL;
31 LODL:        a:=ir + sp;                                #1000 = LODL
32              mar:=a; rd; goto contLODL;
33 STOL:        a:=ir + sp;                                #1001 = STOL
34              mar:=a; mbr:=ac; wr; goto contWR;
35 L1010or1011: alu:=tir; if n then goto SUBL;
36 ADDL:        a:=ir + sp;                                #1010 = ADDL
37              mar:=a; rd; goto contADDL;
38 SUBL:        a:=ir + sp;                                #1011 = SUBL
39              mar:=a; rd; goto contSUBL ;
40 L110or111:   tir:=lshift(tir); if n then goto L1110or1111;
41 L1100or1101: alu:=tir; if n then goto JNZE;
42 JNEG:        alu:=ac; if n then goto contJUMPS;          #1100 = JNEG
43              goto BEGIN;
44 JNZE:        alu:=ac; if z then goto BEGIN;              #1101 = JNZE
45              pc:=band(ir,amask); goto BEGIN;
46 L1110or1111: tir:=lshift(tir); if n then goto F0orF1;
47 CALL:        sp:=sp + (-1);                              #1110 = CALL
48              mar:=sp; mbr:=pc; wr;
49              pc:=band(ir,amask); wr; goto BEGIN;
50 F0orF1:      tir:=lshift(tir); if n then goto F10orF11;
51 F00orF01:    tir:=lshift(tir); if n then goto F010orF011;
52 F000orF001: alu:=tir; if n then goto POPI;
53 PSHI:        mar:=ac; rd;                                #1111-0000 = PSHI

```

```

54         sp:=sp + (-1); rd;
55         mar:=sp; wr; goto contWR;
56 POPI:   mar:=sp; sp:=sp + 1; rd;           #1111-0010 = POPI
57         rd;
58         mar:=ac; wr; goto contWR;
59 F010orF011: alu:=tir; if n then goto POP;
60 PUSH:   sp:=sp + (-1);                   #1111-0100 = PUSH
61         mar:=sp; mbr:=ac; wr; goto contWR;
62 POP:    mar:=sp; sp:=sp + 1; rd;         #1111-0110 = POP
63         rd;
64         ac:=mbr; goto BEGIN;
65 F10orF11: tir:=lshift(tir); if n then goto F110orF111;
66 F100orF101: alu:=tir; if n then goto SWAP;
67 RETN:   mar:=sp; sp:=sp + 1; rd;         #1111-1000 = RETN
68         rd;
69         pc:=mbr; goto BEGIN;
70 SWAP:   a:=ac;                           #1111-1010 = SWAP
71         ac:=sp;
72         sp:=a; goto BEGIN;
73 F110orF111: tir:=lshift(tir); if n then goto F1110orF1111;
74 INSP:   a:=band(ir,smask);               #1111-1100 = INSP
75 contDESP: sp:=sp + a; goto BEGIN;
76 F1110orF1111: alu:=tir; if n then goto HALT; #HALT or DESP?
77 DESP:   a:=band(ir, smask);             #1111-1110 = DESP
78         a:=inv(a);
79         a:=a + 1; goto contDESP;

```

By the way, note the use of labels here. For example, BEGIN refers to location 0 in the control store, L0or1 refers to location 2,⁵ and so on. Of course, the assembler will make the conversion, as with other assemblers.

Let's take a look at some of the code. For any MAC-1 instruction, the first few MIC-1 instructions will be devoted to fetching and decoding the MAC-1 instruction. The fetching is seen in the first two MIC-1 instructions:

```

BEGIN:    mar:=pc; rd;
          pc:=pc + 1; rd;

```

The rd must be asserted for two clock cycles, as that is the time needed to access MAC-1'S memory. Then the (MAC-1) instruction decode begins. By the time we get to the third MIC-1 instruction,

```

L0or1:   ir:=mbr; if n then goto L10or11;

```

⁵These numbers, 0, 1, 2, ... are displayed here, to remind you which MIC-1 memory locations the microinstructions will be stored in. They are not line numbers (though it is convenient to call them such), and the assembler will not tolerate their presence in source code.

the MAC-1 instruction has been fetched, and is now in mbr.⁶ In executing `ir:=mbr`, the contents of the mbr will pass through the ALU, thus affecting the n and z bits. So, in executing the `if n then goto L10or11`, what we are really doing is saying, “If bit 15 of the MAC-1 instruction is a 1, then goto L10or11.” Then at L10or11, we have the MIC-1 instruction

```
tir:=lshift(ir + ir); if n then goto L110or111;
```

Again this MIC-1 instruction has the goal of testing certain bits within the MAC-1 instruction. Look very, very carefully at how this is done. You see an explicit shift there, in the form of the mal mnemonic `lshift` (recall that the mal assembler will translate this to a 10 in the `sh` field). But there also is an implicit shift: the operation `ir+ir` is equivalent to multiplying `ir` by 2, which in turn is equivalent to a left shift of one bit. So, there are actually two left shifts. On the other hand, though, the `n` bit will only reflect the first of these (the `ir+ir`), because it is the only one to have been done by the ALU. So, the end result is that we are testing bit 14 of the MAC-1 instruction, not bit 13. (But the second left shift is done in preparation for testing the latter bit.)

It is crucial that you go through the entire microcode for a few MAC-1 instructions, including at least one that accesses the stack. Make absolutely sure that you understand all aspects.

⁶Review from ECS 50/EEC 70: the memory data register/memory buffer register and memory address register in a CPU are the CPU's interface to the data and address buses, respectively. This is how the CPU transfers data to and from memory.

.1 MIC-1 Block Diagram

