Introduction to Microcoded Implementation of a CPU Architecture

N.S. Matloff, revised by D. Franklin

January 30, 1999, revised March 2004

1 Microcoding

Throughout the years, Microcoding has changed dramatically. The debate over simple computers vs complex computers once raged within the architecture community. In the end, the most popular microcoded computers survived for three reasons - marketshare, technological improvements, and the embracing of the principles used in simple computers. So the two eventually merged into one. To truly understand microcoding, one must understand why they were built, what they are, why they survived, and, finally, what they look like today.

1.1 Motivation

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 languge program, say a .EXE file in the Intel/DOS case, would run on any CPU in the family.

When desigining an instruction set architecture, there is a tradeoff between software and hardware. If you provide very few instructions, it takes more instructions to perform the same task, but the hardware can be very simple. If you provide a rich set of instructions, it takes fewer instructions to execute the same task, but a purely hardware implementation can get prohibitively complex.

Why are fewer instructions, or a smaller overall program, desirable? Instructions are stored in memory, and memory can be incredibly slow, as you will learn when we reach the caches section of the class. At one time, all memory operations took several cycles, so the more work a single instruction could perform, the better.

On the other hand, simple hardware is desired. Simple hardware has two large advantages. First, they are easy to design and verify, which means quicker time-to-market and lower design costs. In addition, the chips will be smaller, making them cheaper to manufacture.

The ideal would be a complex instruction set (requiring fewer instructions to run the same program) with simple harware. The two do not match, so a compromise was struck. Create a program that

runs strictly on the hardware, stored in the hardware, that takes complex instructions and breaks them down into simple instructions for the simple hardware. This is an attempt to get the best of both worlds - complex instruction set with simple hardware.

1.2 Hardwired Versus Microcoded Implementation

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 requires more chip space. Larger chips are more expensive, but the larger control store will net better performance.

The execution of an instruction in the target architecture occurs via the execution of several instructions (*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 (Read Only Memory), and fetching from it slows down the show. Plus, decoding the instruction takes several cycles, whereas hard-coded decoding takes only one cycle.

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. The problem with users adding machine-level instructions is that assemblers would not recognize the new instruction, nor would compilers generate it. Instead, 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. This library approach is often used in the DSP (Digital Signal Processing) market because manufacturers have added special instructions for specific algorithms (like FFT), and the compiler is not intelligent enough to figure out when to use them.

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 the diagram, thin lines are control lines, and thick lines are data lines (both result computation and NextPC).

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.

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

2.1.2 Micro-code control

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 may include a branch, and a condition field indicating whether or not the branch will actually be taken. In MIC-1, the corresponding fields are cond (when to branch) and addr (where to branch). 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.

2.1.3 ALU operations

The ALU can perform four operations: a+b, a & b, a, and a. It may seem odd to include the operation 'a', which merely passes the first value through. Sometimes you want to inspect a value without changing it (using the n and z bits, described below). This is also necessary when writing to the mbr in a store memory operation.

The n and z bits 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.

Note that the shifter is after the ALU. You may shift the result of an ALU operation one bit to the left or right. This occurs after the ALU operation, so it is not considered when determining the n and z bits.

2.1.4 Memory operations

Note that in the diagram, mar (Memory Address Register) and mbr (Memory Buffer Register, i.e. sometimes called Memory Data Register) will be connected to the address and data pins of the

chip, respectively.

Say the CPU wishes to read from memory location 200. It places the 200 in the mar, and asserts a Read line in the control section of the system bus (activated by rd in MIC-1, as we will see later). Since the mar is connected to the address portion of the system bus, 200 will go out onto those bus lines and will be seen by memory. Some time later (whatever the gate delay is for the memory, 2 cycles in the case of of MIC-1) the memory will place the contents of 200, say 12, onto the data portion of the bus. Since the mbr is connected to that portion of the bus, the mbr will now contain 12 and will be available for internal use by the CPU.

Say the CPU wishes to *write* something to location 200, say 5. Then the CPU will place 200 in the mar and 5 in the mbr, from which the 5 will flow out onto the data lines of the bus, and so on.

Here are a few things to remember when using memory in MIC-1:

- To read from memory, write the address to mar (mar := a) and assert the read line for *two* instructions (rd). The data *must* be read from *mbr* in the third cycle (b := mbr); See lines 12-14 of the given microcode implementation for MAC-1 for an example.
- To write to memory, write the address to mar (mar := a) and the data to mbr (mbr := b) in the *same instruction*, and assert the write line for *two* instructions (wr). See lines 9-10 of the given microcode implementation for MAC-1 for an example.
- The physical address space in MIC-1 is only 12 bits wide, whereas the data path is 16 bits wide. This means that when you write into mar, only the lower 12 bits of the value are used. The upper bits do not matter at all.
- There is no physical connection from the output of the alu to mar, so you can not place the result of an alu operation into mar.

2.1.5 Register file

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.

Similarly, since MAC-1 has 12-bit addresses, Tanenbaum set MIC-1's pc, sp and mar registers to be 12 bits wide. All of the other registers are 16 bits wide. You must be careful with pc, ac, and sp. At the end of each macrocode instruction, they must be the correct value. If ac is not specified as changing in that macrocode operation, its value must be the same as in the beginning. There are several registers that are hard-coded for your convenience since there is no "addi" instruction with which to bring constants into the microcode. Finally, registers a-f may be used for anything you wish.

2.1.6 Execution on MIC-1

Here are the steps to execution in mic-1:

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.
3. The amux is used to decide between the data from the mbr (used when loading from memory) and the a bus. The ALU then computes a result, n, and z bits, and that result is put through the shifter. In addition, if memory is being initiated, the mar takes the value from the b bus.
4. Store ALU output to mbr (for a store) and/or destination register. The mmux takes the n,z, and cond bits and decides between the next mpc (mpc+1) and the addr field in the instruction.

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. Below is a summary of the fields in the instruction format.

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. These would be loosely analogous to rs, rt, and rd fields in MIPS, except that 'b' is *always* used as a source register, never a destination register.

The c field could have been 16 bits wide (holding four register numbers), allowing multiple registers to be written 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*.[2]

[2]The Intel Pentium, for instance, is very horizontal, with 118 bits per microinstruction.

```
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
------ MIC-1 instruction format - (analagous DLX control signals) -----
   amux (bit 31) (MemToReg)            cond (bits 30-29) (Branch,Jump)
   -------------                       -----------------
   0 = input from 'a' latch            00 = no branch
   1 = input from mbr                  01 = branch if n = 1
                                       10 = branch if z = 1
                                       11 = unconditional branch

   alu (bits 28-27) (ALUOp)            sh (bits 26-25) (shamt field)
   ----------------                    ---------------
   00 = a + b                          00 = no shift
   01 = a and b                        01 = shift right 1 bit
   10 = a                              10 = shift left 1 bit
   11 = not a                          11 = (not used)

   mbr (bit 24)                        mar (bit 23)
   ------------                        ------------
   1 = load mbr from shifter           1 = load mar from B latch
   0 = do not load mbr                 0 = do not load mar

   rd,wr (bits 22-21)(MemRd,MemWr)     enc (bit 20) (RegWrite)
   -----------------------             -----------------------
   1 = yes, 0 = no                     1 = yes, 0 = no

   c,b,a (bits 19-8)                   addr (bits 7-0)
   -----------------                   --------------
   4-bit register number               branch target
```

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.[3]

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.

One obvious question is, what can I do in a single microcode instruction? The key is, if you can encode it, you can run it. You can figure it out in one of two ways - look at the diagram at the back of the handout and see if all of the things you want to do have datapaths on the computer that do not conflict with each other. You can also look at the encoding bits and see if you can set the bits such that all that you want to get done gets done.

Look at the diagram and figure out whether or not the following instructions are legal (the answers are given near the end of the handout). Note that each instruction has several statements in it. Semi-colons do not separate instructions. Only linebreaks separate instructions.

- mar := sp; mbr:=ac; wr; goto contWR;
- mar := a+c; rd;
- mar := sp; rd; b := sp+1;
- mar := a; mbr := a+c; d := a+c; wr; if n then goto F110orF11;
- ac := mbr; a := sp+1
- ac := mbr + a;

[3]Make 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.

2.3 MAC-1

MAC-1 is an accumulator machine. It has a program counter PC, a stack pointer SP, and an accumulator AC. Addressing modes are direct, indirect and local. Instructions are summarized below, all of which are 16 bits.

Local addressing, which is common in many architectures, is a stack-relative mode intended for accessing local variables. An example is given on the next page that shows the allocation and use of local variables in MAC code.

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 is *word addressable*.

```
machine language   mnemonic   long name         action
---------------    --------   ---------         ------
0000xxxxxxxxxxx    LODD       load direct       ac:=m[x]
0001xxxxxxxxxxx    STOD       store direct      m[x]:=ac
0010xxxxxxxxxxx    ADDD       add direct        ac:=ac+m[x]
0011xxxxxxxxxxx    SUBD       subtract direct   ac:=ac-m[x]
0100xxxxxxxxxxx    JPOS       jump positive     if ac >= 0 then pc:=x
0101xxxxxxxxxxx    JZER       jump zero         if ac = 0 then pc:=x
0110xxxxxxxxxxx    JUMP       jump              pc:=x
0111xxxxxxxxxxx    LOCO       load constant     ac:=x, x unsigned
1000xxxxxxxxxxx    LODL       load local        ac:=m[sp+x]
1001xxxxxxxxxxx    STOL       store local       m[x+sp]:=ac
1010xxxxxxxxxxx    ADDL       add local         ac:=ac+m[sp+x]
1011xxxxxxxxxxx    SUBL       subtract local    ac:=ac-m[sp+x]
1100xxxxxxxxxxx    JNEG       jump negative     if ac < 0 then pc:=x
1101xxxxxxxxxxx    JNZE       jump nonzero      if ac != 0 then pc:=x
1110xxxxxxxxxxx    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
```

```
--------------- C code to MAC translation example ----------------------
int SumToN(int N){          DESP 2          sp := sp - 2
   int i,sum;               LOCO 0          ac := 0
   sum = 0;                 STOL 0          m[sp+0] := ac (sum := 0)
   i = 0;                   STOL 1          m[sp+1] := ac (i := 0)
   while (i < num)     5:   SUBL 2          ac := ac - m[sp+3] (i - N)
   {                        JZER 15         jump if (i == N)
                            JPOS 15         jump if (i > N)
        sum = sum + i;      LODL 0          ac := m[sp+0] (sum)
                            ADDL 1          ac := ac + m[sp+1] (sum + i)
                            STOL 0          m[sp+0] := ac  (sum = sum + i)
        i++;                LOCO 1          ac := 1
                            ADDL 1          ac := ac + m[sp+1] (1 + i)
                            STOL 1          m[sp+1] := ac (i = 1+i)
   }                        JUMP 5
                       15:  LODL 0          ac := m[sp+0] (sum)
   return sum;              INSP 2          sp := sp + 2
}                           STOL 1          m[sp+1] := sum (return sum)
                            RETN            return to caller
```

2.3.1 MIC-1 Implementation of MAC-1

Here is the microcode for the implementation: (Each line is a **single** instruction)

```
 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,[4] 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;
```

the MAC-1 instruction has been fetched, and is now in mbr. [5] 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 go to 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.

Solutions to legal vs illegal questions:

- mar := sp; mbr:=ac; wr; goto contWR; *legal - amux=0, cond=11, alu=10, sh=00, mbr=1, mar=1, wr=1,rd=0,enc=0, c=dc, a=ac, b=sp, addr=contWR*
- mar := a+c; rd; *illegal - mar can not use the result of an alu operation*

[4] These 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.

[5] 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.

- mar := sp; rd; b := sp+1; *legal - amux=0, cond=00, alu=00, sh=00, mbr=0, mar=1, wr=0,rd=1,enc=1, c=b, a=1, b=sp, addr=dc*
- mar := a; mbr := a+c; d := a+c; wr; if n then goto F110orF111; *legal - amux=0, cond=01, alu=00, sh=00, mbr=1, mar=1, wr=1,rd=0,enc=1, c=d, a=c, b=a, addr=F110orF111*
- ac := mbr; a := sp+1 *illegal - we can not write to both ac and a in the same instruction*
- ac := mbr + a; *legal - amux=1, cond=00, alu=00, sh=00, mbr=0, mar=0, wr=0,rd=0,enc=1, c=ac, a=dc, b=a, addr=dc*

3 Survival of the Fittest - Microcoding Evolution

As you can see from the microcode, to execute a single instruction, it takes several cycles to fetch, decoded, and then finally execute it. This can be slower than the several instructions that a simple instruction set would have provided! In fact, two factors made RISC chips much more desirable. First, a large boost in memory performance provided by a cache (covered later in the course) meant that instructions were no longer incredibly slow to fetch. In addition, increasingly complex chips allowed simple instructions to be executed out of order, which was more difficult to do with complex instructions. The complex instruction set itself inhibited both compiler and hardware optimizations. This could have spelled the demise of the microcoded computer.

Two factors contributed to the survival of the Intel x86 family: market factors and technology. Once enough companies have invested in software for a certain instruction set architecture, companies are unlikely to change architectures. They would need to buy all new software when they upgraded to a new chip. Investing in both hardware upgrades and completely new software at the same time can be prohibitively expensive.

The larger factor was that advances in fabrication technology allowed much more logic to be placed on a single chip. In the architecture presented above, each macrocode instruction is executed start to finish with a sequence of microcode instructions. The increased real estate allowed for a vastly different implementation. Macrocode instructions are fetched and decoded in hardware. This hardware translates the macroinstruction into a set of microcode instructions that implements the macro instruction. This hardware translation is faster than the fetch and decode done in microcode. In addition, this set of microcode instructions can be fed into a simple computer core which performs out of order execution with all of the hardware optimizations allowed in simple cores. Again, microcoded computers try to get the best of both worlds - the small code size in order to decrease the number of instructions fetched, but a RISC internal implementation to take advantage of hardware optimizations. The only advantage lost is the fact that the compiler can not perform optimizations on microcode - it is generated at run-time by hardware translation.

Where does the future of microcoded computers lie? Intel has developed an alternative to its x86 Pentium family - the Itanium. This is closer to a RISC instruction set than CISC, with the compiler packaging parallel instructions into larger chunks of instructions. As compiler technology matures, it is becoming a larger part of program performance, pulling architectures once again to RISC.

.1 MIC-1 Block Diagram