4.2. AN EXAMPLE MICROARCHITECTURE

Now that we have reviewed all the basic components from which the microprogramming level is constructed, it is time to see how they are connected. Because general principles are few and far between in this area, we will introduce the subject by means of a detailed example.

4.2.1. The Data Path

The **data path** of our example microarchitecture is shown in Fig. 4-8. (The data path is that part of the CPU containing the ALU, its inputs, and its outputs.) It contains 16 identical 16-bit registers, labeled PC, AC, SP, and so on, that form a scratchpad memory accessible only to the microprogramming level. The registers labeled 0, +1, and −1 will be used to hold the indicated constants; the meaning of the other register names will be explained later. (Actually, 0 is never used in our simple examples but it probably would be needed in a more complicated machine, so we have included it because we have more registers than we can use anyway.) Each register can output its contents onto one or both of two internal buses, the A bus and the B bus, and each can be loaded from a third internal bus, the C bus, as shown in the figure.

The A and B buses feed into a 16-bit-wide ALU that can perform four

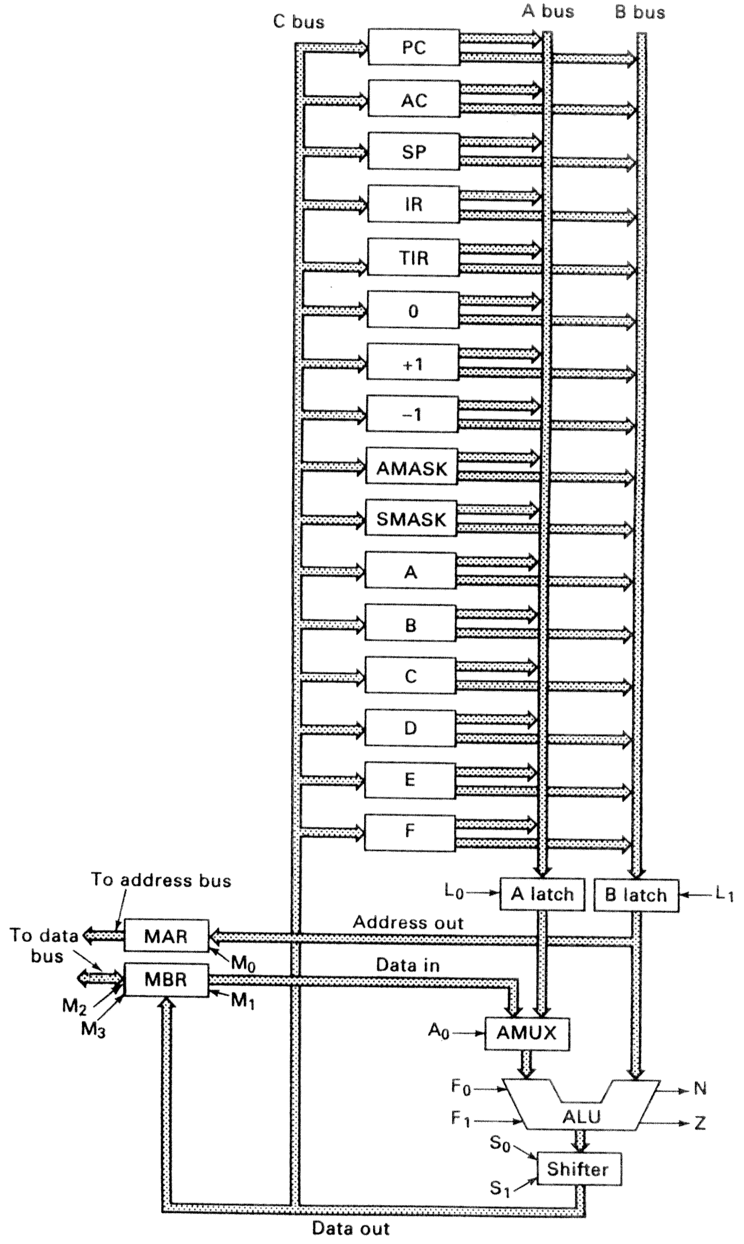


Fig. 4-8. The data path of the example microarchitecture used in this chapter.

functions: A + B, A AND B, A, and NOT A. The function to be performed is specified by the two ALU control lines, F_0 and F_1. The ALU generates two status bits based on the current ALU output: N, which is set when the ALU output is negative, and Z, which is set when the ALU output is zero.

The ALU output goes into a shifter, which can shift it 1 bit in either direction, or not at all. It is possible to perform a 2-bit left shift of a register, R, by computing R + R in the ALU (which is a 1-bit left shift), and then shifting the sum another bit left using the shifter.

Neither the A bus nor the B bus feeds the ALU directly. Instead, each one feeds a latch (i.e., a register) that in turn feeds the ALU. The latches are needed because the ALU is a combinational circuit—it continuously computes the output for the current input and function code. This organization can cause problems when computing, for example, A := A + B. As A is being stored into, the value on the A bus begins to change, which causes the ALU output and thus the C bus to change as well. Consequently, the wrong value may be stored into A. In other words, in the assignment A := A + B, the A on the right-hand side is the original A value, not some bit-by-bit mixture of the old and new values. By inserting latches in the A and B buses, we can freeze the original A and B values there early in the cycle, so the ALU is shielded from changes on the buses as a new value is being stored in the scratchpad. The loading of the latches is controlled by L_0 and L_1.

It is worth pointing out that our solution to this problem (i.e., inserting latches in front of the ALU) is not the only one. If all the registers are flip-flops rather than latches, then a two-bus design is also feasible by loading the operands onto the A and B buses early in the cycle and reading the result from one of the buses late in the cycle. The trade-offs between two and three bus designs involve complexity, parallelism, and amount of wiring. A more detailed treatment of these issues is beyond the scope of this book.

To communicate with memory, we have included an MAR and an MBR in the microarchitecture. The MAR can be loaded from the B latch, in parallel with an ALU operation. The M_0 line controls loading of MAR. On writes, the MBR can be loaded with the shifter output, in parallel with, or instead of, a store back into the scratchpad. M_1 controls loading MBR from the shifter output. M_2 and M_3 control reads and writes from memory. On reads, the data read from memory can be presented to the left input of the ALU via the A multiplexer, indicated by Amux in Fig. 4-8. A control line, A_0, determines whether the A latch or the MBR is fed into the ALU. The microarchitecture of Fig. 4-8 is similar to that of several commercially available bit slices.

4.2.2. Microinstructions

To control the data path of Fig. 4-8 we need 61 signals. These can be divided into nine functional groups, as described below.

16 signals to control loading the A bus from the scratchpad

16 signals to control loading the B bus from the scratchpad

16 signals to control loading the scratchpad from the C bus

2 signals to control the A and B latches

2 signals to control the ALU function

2 signals to control the shifter

4 signals to control the MAR and MBR

2 signals to indicate memory read and memory write

1 signal to control the Amux

Given the values of the 61 signals, we can perform one cycle of the data path. A cycle consists of gating values onto the A and B buses, latching them in the two bus latches, running the values through the ALU and shifter, and finally storing the results in the scratchpad and/or MBR. In addition, the MAR can also be loaded, and a memory cycle initiated. As a first approximation, we could have a 61-bit control register, with one bit for each control signal. A 1 bit means that the signal is asserted and a 0 means that it is negated.

However, at the price of a small increase in circuitry, we can greatly reduce the number of bits needed to control the data path. To begin with, we have 16 bits for controlling input to the A bus, which allows 2^{16} combinations of source registers. Only 16 of these combinations are permitted—namely, each of the 16 registers all by itself. Therefore, we can encode the A bus information in 4 bits and use a decoder to generate the 16 control signals. The same holds for the B bus.

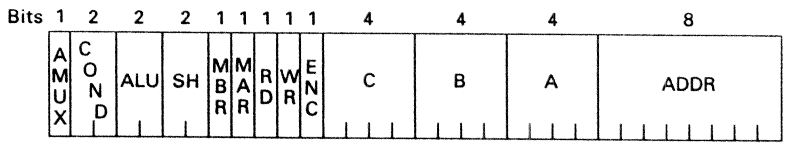
The situation is slightly different for the C bus. In principle, multiple simultaneous stores into the scratchpad are feasible, but in practice this feature is virtually never useful and most hardware does not provide for it. Therefore, we will also encode the C bus control in 4 bits. Having saved $3 \times 12 = 36$ bits, we now need 25 control bits to drive the data path. L_0 and L_1 are always needed at a certain point in time, so they will be supplied by the clock, leaving us with 23 control bits. One additional signal that is not strictly required, but is often useful, is one to enable/disable storing the C bus in the scratchpad. In some situations one merely wishes to perform an ALU operation to generate the N and Z signals but does not wish to store the result. With this extra bit, which we will call ENC (ENable C), we can indicate that the C bus is to be stored (ENC = 1) or not (ENC = 0).

At this point we can control the data path with a 24-bit number. Now we note that RD and WR can be used to control latching MBR from the system data bus and enabling MBR onto it, respectively. This observation reduces the number of independent control signals from 24 down to 22.

The next step in the design of the microarchitecture is to invent a microinstruction format containing 22 bits. Figure 4-9 shows such a format, with two additional fields COND and ADDR, which will be described shortly. The microinstruction contains 13 fields, 11 of which are as follows:

AMUX	— controls left ALU input: 0 = A latch, 1 = MBR
ALU	— ALU function: 0 = A + B, 1 = A AND B, 2 = A, 3 = $\bar{A}$
SH	— shifter function: 0 = no shift, 1 = right, 2 = left
MBR	— loads MBR from shifter: 0 = don't load MBR, 1 = load MBR
MAR	— loads MAR from B latch: 0 = don't load MAR, 1 = load MAR
RD	— requests memory read: 0 = no read, 1 = load MBR from memory
WR	— requests memory write: 0 = no write, 1 = write MBR to memory
ENC	— controls storing into scratchpad: 0 = don't store, 1 = store
C	— selects register for storing into if ENC = 1: 0 = PC, 1 = AC, etc.
B	— selects B bus source: 0 = PC, 1 = AC, etc.
A	— selects A bus source: 0 = PC, 1 = AC, etc.

The ordering of the fields is completely arbitrary. This ordering has been chosen to minimize line crossings in a subsequent figure. (Actually, this criterion is not as crazy as it sounds; line crossings in figures usually correspond to wire crossings on printed circuit boards or on chips, which cause trouble in two-dimensional designs.)



AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch	0 = No jump	0 = A + B	0 = No shift	0 = No
1 = MBR	1 = Jump if N = 1	1 = A AND B	1 = Shift right 1 bit	1 = Yes
	2 = Jump if Z = 1	2 = A	2 = Shift left 1 bit	
	3 = Jump always	3 = $\bar{A}$	3 = (not used)	

Fig. 4-9. The microinstruction layout for controlling the data path of Fig. 4-8.

4.2.3. Microinstruction Timing

Although our discussion of how a microinstruction can control the data path during one cycle is almost complete, we have neglected one issue up until now: timing. A basic ALU cycle consists of setting up the A and B latches, giving the

ALU and shifter time to do their work, and storing the results. It is obvious that these events must happen in that sequence. If we try to store the C bus into the scratchpad before the A and B latches have been loaded, garbage will be stored instead of useful data. To achieve the correct event sequencing, we now introduce a four-phase clock, that is, a clock with four subcycles, like that of Fig. 4-5. The key events during each of the four subcycles are as follows:

1. Load the next microinstruction to be executed into a register called **MIR**, the MicroInstruction Register.
2. Gate registers onto the A and B buses and capture them in the A and B latches.
3. Now that the inputs are stable, give the ALU and shifter time to produce a stable output and load the MAR if required.
4. Now that the shifter output is stable, store the C bus in the scratchpad and load the MBR, if either is required.

Figure 4-10 is a detailed block diagram of the complete microarchitecture of our example machine. It may look imposing initially but it is worth studying carefully. When you fully understand every box and every line on it, you will be well on your way to understanding the microprogramming level. The block diagram has two parts, the data path, on the left, which we have already discussed in detail, and the control section, on the right, which we will now look at.

The largest and most important item in the control portion of the machine is the **control store**. This special, high-speed memory is where the microinstructions are kept. On some machines it is read-only memory; on others it is read/write memory. In our example, microinstructions will be 32 bits wide and the microinstruction address space will consist of 256 words, so the control store will occupy a maximum of $256 \times 32 = 8192$ bits.

Like any other memory, the control store needs an MAR and an MBR. We will call the MAR the **MPC** (MicroProgram Counter), because its only function is to point to the next microinstruction to be executed. The MBR is just the **MIR** as mentioned above. Be sure you realize that the control store and main memory are completely different, the former holding the microprogram and the latter the conventional machine language program.

From Fig. 4-10 it is clear that the control store continuously tries to copy the microinstruction addressed by the MPC into the MIR. However, the MIR is only loaded during subcyle 1, as indicated by the dashed line from the clock to it. During the other three subcycles it is not affected, no matter what happens to MPC.

During subcycle 2, the MIR is stable, and the various fields begin controlling the data path. In particular, A and B gate onto the A and B buses. The A decoder

0 = Do not jump; next microinstruction is taken from MPC + 1

1 = Jump to ADDR if N = 1

2 = Jump to ADDR if Z = 1

3 = Jump to ADDR unconditionally

The Micro sequencing logic combines the two ALU bits, N and Z, and the two COND bits, call them L and R for Left and Right, to generate an output. The correct signal is

$$\text{Mmux} = \bar{L}RN + L\bar{R}Z + LR = RN + LZ + LR$$

where + means INCLUSIVE OR. In words, the control signal to Mmux is 1 (routing ADDR to MPC) if LR is 01_2 and N = 1, or LR is 10_2 and Z = 1 or LR is 11_2. Otherwise, it is 0 and the next microinstruction in sequence is fetched. The circuit to compute the Mmux signal can be built from SSI components, as in Fig. 3-3(b), or be part of a PLA, as in Fig. 3-16.

To make our example machine slightly realistic, we will assume that a main memory cycle takes longer than a microinstruction. In particular, if a microinstruction starts a main memory read, by setting RD to 1, it must also have RD = 1 in the next microinstruction executed (which may or may not be located at the next control store address). The data become available two microinstructions after the read was initiated. If the microprogram has nothing else useful to do in the microinstruction following the one that initiated a memory read, the microinstruction just has RD = 1 and is effectively wasted. In the same way, a memory write also takes two microinstruction times to complete.

4.3. AN EXAMPLE MACROARCHITECTURE

To continue our microprogramming level example, we now switch to the architecture of the conventional machine level to be supported by the interpreter running on the machine of Fig. 4-10. For convenience, we will call the architecture of the level 2 or 3 machine the **macroarchitecture**, to contrast it with level 1, the microarchitecture. (For the purposes of this chapter we will ignore level 3 because its instructions are largely those of level 2 and the differences are not important here.) Similarly, the level 2 instructions will be called **macroinstructions**. Thus for the duration of this chapter, the normal ADD, MOVE, and other instructions of the conventional machine level will be called macroinstructions. (The point of repeating this remark is that some assemblers have a facility to define assembly-time "macros," which are in no way related to what we mean by

macroinstructions.) We will sometimes refer to our example level 1 machine as Mic-1 and the level 2 machine as Mac-1. Before we describe Mac-1, however, we will digress slightly to motivate its design.

4.3.1. Stacks

A modern macroarchitecture should be designed with the needs of high-level languages in mind. One of the most important design issues is addressing. To illustrate the problem that must be solved, consider the Pascal program of Fig. 4-11(a). The main program initializes two vectors, x and y, with values such that $x_k = k$ and $y_k = 2k + 1$. Then it computes the inner product (also called a dot product) of the two vectors. Whenever it needs to multiply two small positive integers, it calls the function *pmul*. (Imagine that the compiler is for a microcomputer and only implements a subset of Pascal, not including the multiplication operator.)

Block-structured languages like Pascal are normally implemented in such a way that when a procedure or function is exited, the storage it had been using for local variables is released. The easiest way to achieve this goal is by using a data structure called a stack. A **stack** is a contiguous block of memory containing some data and a **stack pointer** (SP) telling where the top of the stack is. The bottom of the stack is at a fixed address and will not concern us further. Figure 4-12(a) depicts a stack occupying six words of memory. The bottom of the stack is at 4020 and the top of the stack, where SP points, is at 4015. Our stacks will grow from high memory addresses to low ones but the opposite choice is equally good.

Several operations are defined on stacks. Two of the most important are PUSH X and POP Y. PUSH advances the stack pointer (by decrementing it in our example) and then puts X into memory at the location now pointed to by SP. PUSH increases the stack size by one item. POP Y, in contrast, reduces the stack size by storing the top item on the stack in Y, and then removing it by incrementing the stack pointer by the size of the item popped. Figure 4-12(b) shows how the stack of Fig. 4-12(a) looks after a word containing 5 has been pushed on the stack.

Another operation that can be performed on a stack is advancing the stack pointer without actually pushing any data. This is normally done when a procedure or function is entered, to reserve space for local variables. Figure 4-13(a) shows how memory might be allocated during the execution of the main program of Fig. 4-11. We have arbitrarily assumed that the memory consists of 4096 16-bit words, and that the words 4021 to 4095 are used by the operating system, and hence not available for storing variables. The Pascal variable k is stored at address 4020. (Addresses are given in decimal.) The array x requires 20 words, from 4000 to 4019. The array y starts at 3980 for $y[1]$ and extends to 3999 for $y[20]$. While the main program is executing outside *pmul*, SP has the value 3980, indicating that the top of the stack is at 3980.

When the main program wants to call *pmul*, it first pushes the parameters of the call, 2 and k, onto the stack, and then executes the call instruction, which pushes the

```
program InnerProduct (output );

{This program initializes two vectors, x and y, of 20 elements each,
 then computes their inner product :
     x[1] * y[1] + x[2] * y[2] + ... + x[20] * y[20] }

const max = 20;                                    {size of the vectors}

type SmallInt = 0..100;
       vec = array[1..max ] of SmallInt ;

var k : integer ;
     x , y : vec ;


function pmul (a , b : SmallInt ): integer ;
{This function multiplies its two parameters together and returns the product.
 It performs the multiplication by repeated addition .}
var p , j : integer ;
begin                                              {0: reserve stack space for p and j}
  if (a = 0) or (b = 0) then                       {1: if either one is 0, result is 0}
     pmul := 0                                     {2: function returns 0}
  else
     begin
       p := 0;                                     {3: initialize p}
       for j := 1 to a do                          {4: add b to p a times}
          p := p + b ;                             {5: do the addition}
       pmul := p                                   {6: assign result to function}
     end
 end; {pmul}                                       {7: remove locals and return value}


procedure inner (var v : vec ; var ans : integer );
{Compute the inner product of v and x and return it in ans.}
var sum , i : integer ;
begin                                              {8: reserve stack space for sum and i}
  sum := 0;                                        {9: sum will accumulate inner product}
  for i := 1 to max do                             {10: loop through all the elements}
     sum := sum + pmul (x [i ], v [i ]);           {11: accumulate one term}
  ans := sum                                       {12: copy result to ans}
end; {inner}                                       {13: remove sum and i and return}


begin                                              {14: reserve space for k, x, and y}
  for k := 1 to max do                             {15: initialization loop}
     begin
       x [k ] := k ;                               {16: initialize x}
       y [k ] := pmul (2, k ) + 1                  {17: initialize y}
     end;
  inner (y , k );                                  {18: call inner}
  writeln (k )                                     {19: print results}
end.
```

Fig. 4-11(a). A Pascal program to compute an inner product.

```
         K = 4020           /DEFINE SOME SYMBOLS
         X = 4000
         Y = 3980
         A = 4
         B = 3
         P = 1
         J = 0
         V = 5
         ANS = 4
         SUM = 1
         I = 0

         JUMP  MAIN         /START AT MAIN PROGRAM
PMUL:    DESP  2            /0
         LODL  A            /1
         JNZE  ANOTZ        /JUMP IF A <> 0
         LOCO  0            /2
         JUMP  DONE         /RETURN 0
ANOTZ:   LODL  B            /AÇ := B
         JNZE  BNOTZ        /JUMP IF B <> 0
         LOCO  0            /2
         JUMP  DONE         /RETURN 0
BNOTZ:   LOCO  0            /3
         STOL  P            /P := 0
         LOCO  1            /4
         STOL  J            /J := 1
         LODL  A            /CAN LOOP BE EXECUTED?
         JNEG  L2           /A < 0, DO NOT LOOP
         JZER  L2           /A = 0, DO NOT LOOP
L1:      LODL  P            /5
         ADDL  B            /AC := P + B
         STOL  P            /P := P + B
         LOCO  1            /TEST AT END OF LOOP
         ADDL  J            /AC := J + 1
         STOL  J            /J := J + 1
         SUBL  A            /AC := J - A
         JNEG  L1           /JUMP IF J < A
         JZER  L1           /JUMP IF J = A
L2:      LODL  P            /6
DONE:    INSP  2            /7
         RETN               /RETURN

INNER:   DESP  2            /8
         LOCO  0            /9
         STOL  SUM          /SUM := 0
         LOCO  1            /10
         STOL  I            /I := 1
L3:      LOCO  X-1          /11
         ADDL  I            /AC := X + I - 1
         PSHI               /PUSH X[I]
         LODL  V            /AC := ADDRESS OF VECTOR
         ADDL  I            /AC := V + I
         SUBD  C1           /V BEGINS AT 1, NOT 0
         PSHI               /PUSH V[I]
         CALL  PMUL         /PMUL(X[I],V[I])
         INSP  2            /REMOVE PARAMS
         ADDL  SUM          /AC := SUM + PMUL(...)
         STOL  SUM          /SUM := SUM + PMUL(...)
         LOCO  1            /TEST AT END OF LOOP
         ADDL  I            /AC := I + 1
         STOL  I            /I := I + 1
         SUBD  C20          /AC := I - MAX
         JNEG  L3           /JUMP IF I < MAX
         JZER  L3           /JUMP IF I = MAX
         LODL  SUM          /12
         PUSH               /PUSH SUM
         LODL  ANS          /AC := ADDRESS OF ANS
         POPI               /ANS := SUM
         INSP  2            /13
         RETN               /RETURN

MAIN:    DESP  41           /14
         LOCO  1            /15
         STOD  K            /K IS NOT A LOCAL
L4:      LODD  K            /16
         PUSH               /PUSH K ONTO STACK
         LOCO  X-1          /AC := (ADDRESS OF X[1])-1
         ADDD  K            /AC := X + K - 1
         POPI               /X[K] := K
         LOCO  2            /17
         PUSH               /PREPARE PMUL(2,...)
         LODD  K            /PREPARE PMUL(2,K)
         PUSH               /BOTH PARAMS PUSHED
         CALL  PMUL         /PMUL(2,K)
         INSP  2            /REMOVE PARAMETERS
         ADDD  C1           /AC := 2*K + 1
         PUSH               /PREPARE Y[K] := 2*K+1
         LOCO  Y-1          /AC := (ADDRESS OF Y[1])-1
         ADDD  K            /AC := Y + K -1
         POPI               /Y[K] := 2*K+1
         LOCO  1            /TEST AT END OF LOOP
         ADDD  K            /AC := K + 1
         STOD  K            /K := K + 1
         SUBD  C20          /AC := K - MAX
         JNEG  L4           /JUMP IF K < 0
         JZER  L4           /JUMP IF K = MAX
         LOCO  Y            /18
         PUSH               /PUSH ADDRESS OF Y
         LOCO  K            /AC := ADDRESS OF K
         PUSH               /PUSH IT ALSO
         CALL  INNER        /PROCEDURE CALL
         INSP  2            /REMOVE PARAMS
         LODD  K            /19
         PUSH               /PREPARE WRITELN(K)
         CALL  OUTNUM1      /LIBRARY ROUTINE
         INSP  1            /REMOVE PARAM
         CALL  STOP         /END OF JOB

C1:      1                  /CONSTANT 1
C20:     20                 /CONSTANT 20
```

Fig. 4-11(b). *Innerproduct* in assembly language.

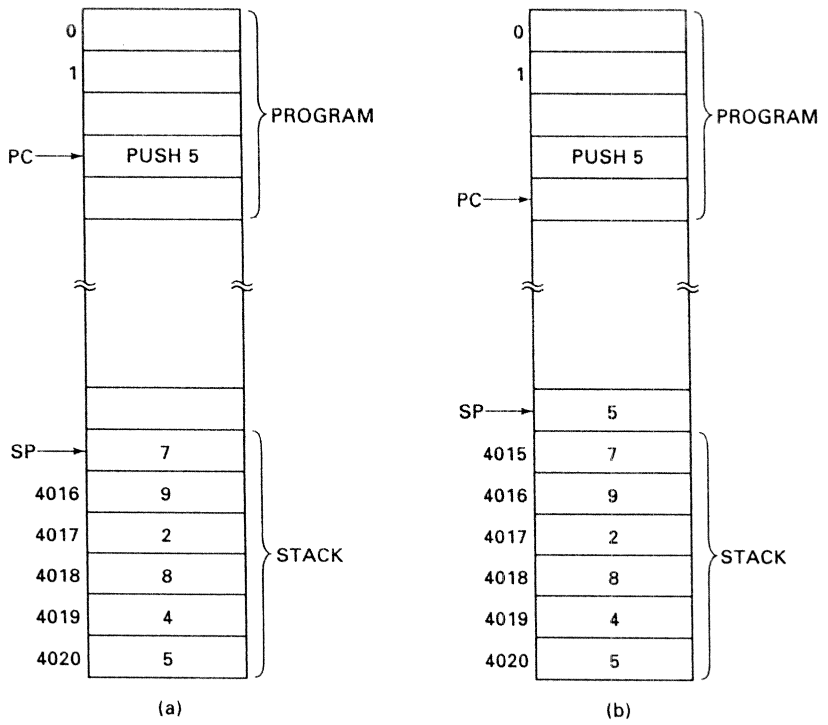


Fig. 4-12. (a) A stack. (b) The stack after pushing 5.

return address onto the stack so that *pmul* will know where to return when it is finished. When *pmul* begins executing, SP is 3977. The first thing it does is advance the stack pointer by 2, to reserve two words for its own local variables, *p* and *j*. At this point SP is 3975, as shown in Fig. 4-13(b). The top five words on the stack constitute the stack frame used by *pmul*; they will be released when it is finished. The words 3979 and 3978 are labeled *a* and *b* because these are the names of *pmul*'s formal parameters but, of course, they contain 2 and *k*, respectively.

When *pmul* has returned and *inner* has been called, the stack configuration is as shown in Fig. 4-13(c). When *inner* calls *pmul*, the stack is as shown in Fig. 4-13(d). Now comes the problem. What code should the compiler generate to access *pmul*'s parameters and locals? If it tries to read *p* using an instruction like MOVE 3976,SOMEWHERE, *pmul* will work when called from the main program but not when called from *inner*. Similarly, MOVE 3971,SOMEWHERE will work when called from *inner*, but not when called from the main program. What is really needed is a way to say "fetch the word 1 higher than the current stack pointer." In other words, the Mac-1 needs an addressing mode that fetches or stores a word at a known distance relative to the stack pointer (or some equivalent addressing mode).

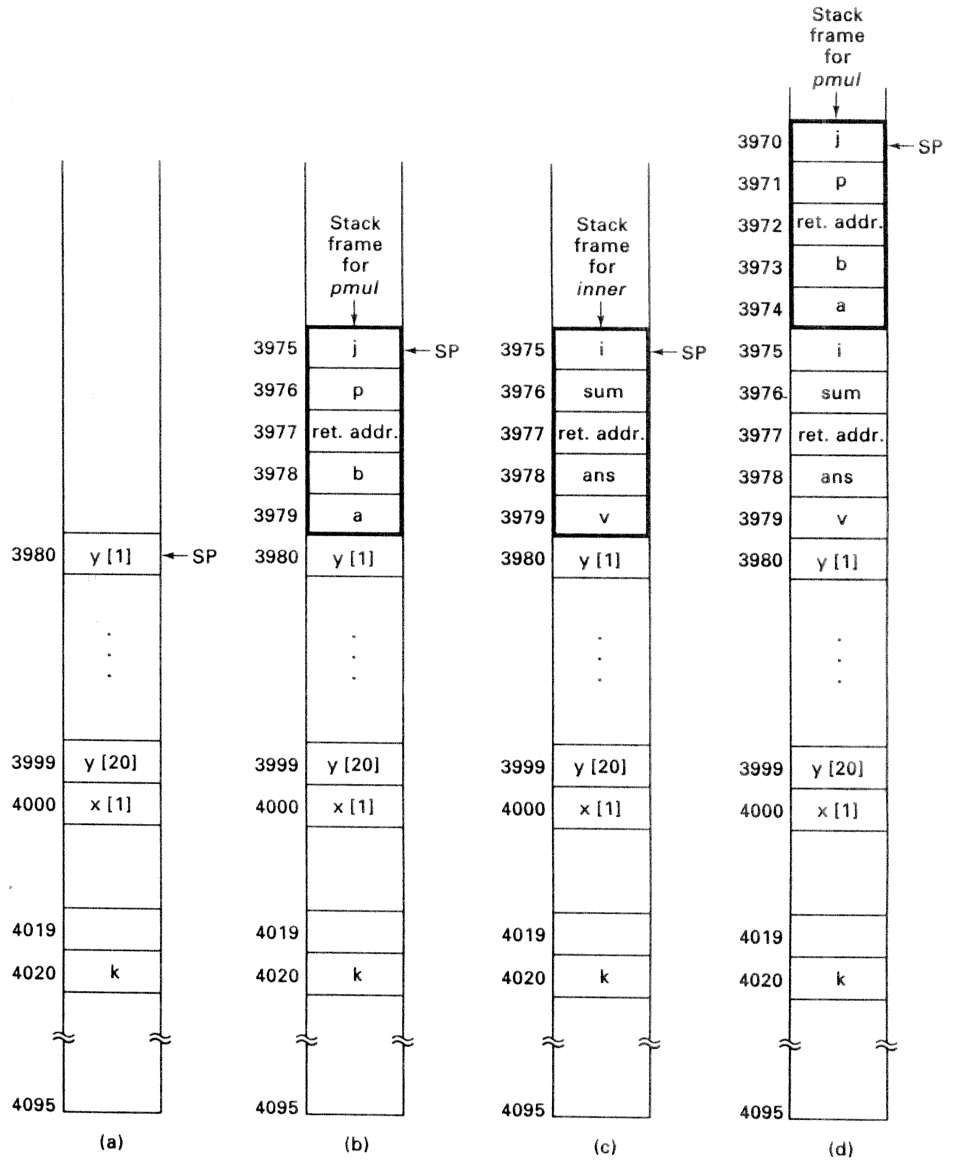


Fig. 4-13. Snapshots of memory during the execution of *InnerProduct*. (a) Stack during execution of the main program. (b) Stack during execution of *pmul*. (c) Stack during execution of *inner*. (d) Stack during execution of *pmul* when called from *inner*.

4.3.2. The Macroinstruction Set

With this addressing mode in mind, we are now ready to look at the Mac-1's architecture. Basically, it consists of a memory with 4096 16-bit words, and three registers visible to the level 2 programmer. The registers are the program counter, PC, the stack pointer, SP, and the accumulator, AC, which is used for moving data around, for arithmetic, and for other purposes. Three addressing modes are provided: direct, indirect, and local. Instructions using direct addressing contain a 12-bit absolute memory address in the low-order 12 bits. Such instructions are useful for accessing global variables, such as x in Fig. 4-11. Indirect addressing allows the programmer to compute a memory address, put it in AC, and then read or write the word addressed. This form of addressing is very general and is used for accessing array elements, among other things. Local addressing specifies an offset from SP and is used to access local variables, as we have just seen. Together, these three modes provide a simple but adequate addressing system.

The Mac-1 instruction set is shown in Fig. 4-14. Each instruction contains an opcode and sometimes a memory address or constant. The first column gives the binary encoding of the instruction. The second gives its assembly language mnemonic. The third gives its name and the fourth describes what it does by giving a Pascal fragment. In these fragments, $m[x]$ refers to memory word x. Thus LODD loads the accumulator from the memory word specified in its low-order 12 bits. LODD is thus direct addressing, whereas LODL loads the accumulator from the word at a distance x above SP, hence is local addressing. LODD, STOD, ADDD, and SUBD perform four basic functions using direct addressing, and LODL, STOL, ADDL, and SUBL perform the same functions using local addressing.

Five jump instructions are provided, one unconditional jump (JUMP) and four conditional ones (JPOS, JZER, JNEG, and JNZE). JUMP always copies its low-order 12 bits into the program counter, whereas the other four only do so if the specified condition is met.

LOCO loads a 12-bit constant in the range 0 to 4095 (inclusive) into AC. PSHI pushes onto the stack the word whose address is present in AC. The inverse operation is POPI, which pops a word from the stack and stores it in the memory word addressed by AC. PUSH and POP are useful for manipulating the stack in a variety of ways. SWAP exchanges the contents of AC and SP, which is useful when SP must be increased or decreased by an amount not known at compile time. It is also useful for initializing SP at the start of execution. INSP and DESP are used to change SP by amounts known at compile time. Due to lack of encoding space, the offsets here have been limited to 8 bits. Finally, CALL calls a procedure, saving the return address on the stack, and RETN returns from a procedure, by popping the return address and putting it into PC.

So far, our machine does not have any input/output instructions. Nor are we about to add any now. It does not need them. Instead, the machine will use memory-mapped I/O. A read from address 4092 will yield a 16-bit word with the next ASCII character from the standard input device in the low-order 7 bits and zeros in the high-order 9 bits. When a character is available in 4092, the high-order bit of the input status register, 4093, will be set. Reading 4092 clears 4093. The input routine will normally sit in a tight loop waiting for 4093 to go negative. When it does, the input routine will load AC from 4092 and return.

Binary	Mnemonic	Instruction	Meaning
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** $ac \geq 0$ **then** $pc := x$
0101xxxxxxxxxxxx	JZER	Jump zero	**if** $ac = 0$ **then** $pc := x$
0110xxxxxxxxxxxx	JUMP	Jump	$pc := x$
0111xxxxxxxxxxxx	LOCO	Load constant	$ac := x$ $(0 \leq x \leq 4095)$
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** $ac \neq 0$ **then** $pc := x$
1110xxxxxxxxxxxx	CALL	Call procedure	$sp := sp - 1; m[sp] := pc; pc := x$
1111000000000000	PSHI	Push indirect	$sp := sp - 1; m[sp] := m[ac]$
1111001000000000	POPI	Pop indirect	$m[ac] := m[sp]; sp := sp + 1$
1111010000000000	PUSH	Push onto stack	$sp := sp - 1; m[sp] := ac$
1111011000000000	POP	Pop from stack	$ac := m[sp]; sp := sp + 1$
1111100000000000	RETN	Return	$pc := m[sp]; sp := sp + 1$
1111101000000000	SWAP	Swap ac, sp	$tmp := ac; ac := sp; sp := tmp$
11111100yyyyyyyy	INSP	Increment sp	$sp := sp + y$ $(0 \leq y \leq 255)$
11111110yyyyyyyy	DESP	Decrement sp	$sp := sp - y$ $(0 \leq y \leq 255)$

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.
yyyyyyy is an 8-bit constant; in column 4 it is called y.

Fig. 4-14. The Mac-1 instruction set.

Output will be done using a similar scheme. A write to address 4094 will take the low-order 7 bits of the word written and copy them to the standard output device. The high-order bit of the output status register, word 4095, will then be cleared, coming back on again when the output device is ready to accept a new character. Standard input and output may be a terminal keyboard and visual display, or a card reader and printer, or some other combination.

As an example of how one programs using this instruction set, see Fig. 4-11(b), which is the program of Fig. 4-11(a) compiled to assembly language by a compiler that does no optimization at all. (Optimized code would make the example hard to follow.) The numbers 0 to 19 in the comments, indicated by a slash in the assembly language, are intended to make it easier to link up the two halves of the figure. OUTNUM1 and STOP are library routines that perform the obvious functions.

4.4. AN EXAMPLE MICROPROGRAM

Having specified both the microarchitecture and the macroarchitecture in detail, the remaining issue is the implementation: What does a program running on the former and interpreting the latter look like, and how does it work? Before we can answer these questions, we must carefully consider in what language we want to do our microprogramming.

4.4.1. The Micro Assembly Language

In principle, we could write microprograms in binary, 32 bits per microinstruction. Masochistic programmers might even enjoy that; certainly nobody else would. Therefore, we need a symbolic language in which to express microprograms. One possible notation is to have the microprogrammer specify one microinstruction per line, naming each nonzero field and its value. For example, to add AC to A and store the result in AC, we could write

ENC = 1, C = 1, B = 1, A = 10

Many microprogramming languages look like this. Nevertheless, this notation is awful and so are they.

A much better idea is to use a high-level language notation, while retaining the basic concept of one source line per microinstruction. Conceivably, one could write microprograms in an ordinary high-level language but because efficiency is crucial in microprograms, we will stick to assembly language, which we define as a symbolic language that has a one-to-one mapping onto machine instructions. Remember that a 25% inefficiency in the microprogram slows the entire machine down by 25%. Let us call our high-level Micro Assembly Language "MAL," which is French for "sick," something you become if you are forced to write too many intricate microprograms for idiosyncratic machines. In MAL, stores into the 16 scratchpad registers or MAR and MBR are denoted by assignment statements. Thus the example in MAL above becomes *ac* := *a* + *ac*. (Because our intention is to make MAL Pascal-like, we will adopt the usual Pascal convention of lowercase italic names for identifiers.)

To indicate the use of the ALU functions 0, 1, 2, and 3, we can write, for example,

ac := *a* + *ac*, *a* := *band*(*ir*, *smask*), *ac* := *a*, and *a* := *inv*(*a*)

respectively, where *band* stands for "Boolean and" and *inv* stands for invert. Shifts can be denoted by the functions *lshift* for left shifts and *rshift* for right shifts, as in

tir := *lshift*(*tir* + *tir*)

which puts *tir* on both the A and B buses, performs an addition, and left shifts the sum 1 bit left before storing it back in *tir*.

Unconditional jumps can be handled with **goto** statements; conditional jumps can test *n* or *z*; for example,

if *n* **then goto** 27

Assignments and jumps can be combined on the same line. However, a slight problem arises if we wish to test a register but not make a store. How do we specify which register is to be tested? To solve this problem we introduce the pseudo variable *alu*, which can be assigned a value just to indicate the ALU contents. For example,

alu := *tir*; **if** *n* **then goto** 27

means *tir* is to be run through the ALU (ALU code = 2) so its high-order bit can be tested. Note that the use of *alu* means that ENC = 0.

To indicate memory reads and writes, we will just put *rd* and *wr* in the source program. The order of the various parts of the source statement is, in principle, arbitrary but to enhance readability we will try to arrange them in the order that they are carried out. Figure 4-15 gives a few examples of MAL statements along with the corresponding microinstructions.

4.4.2. The Example Microprogram

We have finally reached the point where we can put all the pieces together. Figure 4-16 is the microprogram that runs on Mic-1 and interprets Mac-1. It is a surprisingly short program—only 79 lines. By now the choice of names for the scratchpad registers in Fig. 4-8 is obvious: PC, AC, and SP are used to hold the

Statement	AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
mar := *pc*; *rd*	0	0	2	0	0	1	1	0	0	0	0	0	00
rd	0	0	2	0	0	0	1	0	0	0	0	0	00
ir := *mbr*	1	0	2	0	0	0	0	0	1	3	0	0	00
pc := *pc* +1	0	0	0	0	0	0	0	0	1	0	6	0	00
mar := *ir*; *mbr* :=*ac*; *wr*	0	0	2	0	1	1	0	1	0	0	3	1	00
alu :=*tir*; **if** *n* **then goto** 15	0	1	2	0	0	0	0	0	0	0	0	4	15
ac := *inv* (*mbr*)	1	0	3	0	0	0	0	0	1	1	0	0	00
tir := *lshift* (*tir*); **if** *n* **then goto** 25	0	1	2	2	0	0	0	0	1	4	0	4	25
alu := *ac*; **if** *z* **then goto** 22	0	2	2	0	0	0	0	0	0	0	0	1	22
ac := *band* (*ir*, *amask*); **goto** 0	0	3	1	0	0	0	0	0	1	1	8	3	00
sp := *sp* + (−1); *rd*	0	0	0	0	0	0	1	0	1	2	2	7	00
tir := *lshift* (*ir*+ *ir*); **if** *n* **then goto** 69	0	1	0	2	0	0	0	0	1	4	3	3	69

Fig. 4-15. Some MAL statements and the corresponding microinstructions.

three Mac-1 registers. IR is the instruction register and holds the macroinstruction currently being executed. TIR is a temporary copy of IR, used for decoding the opcode. The next three registers hold the indicated constants. AMASK is the address mask, 007777 (octal), and is used to separate out opcode and address bits. SMASK is the stack mask, 000377 (octal), and is used in the INSP and DESP instructions to isolate the 8-bit offset. The remaining six registers have no assigned function and can be used as the microprogrammer wishes.

Like all interpreters, the microprogram of Fig. 4-16 has a main loop that fetches, decodes, and executes instructions from the program being interpreted, in this case, level 2 instructions. Its main loop begins on line 0, where it begins fetching the macroinstruction at PC. While waiting for the instruction to arrive, the microprogram increments PC and continues to assert the RD bus signal. When it arrives, in line 2, it is stored in IR and simultaneously the high-order bit (bit 15) is tested. If bit 15 is a 1, decoding proceeds at line 28; otherwise, it continues on line 3. Assuming for the moment that the instruction is a LODD, bit 14 is tested on line 3, and TIR is loaded with the original instruction shifted left 2 bits, one using the adder and one using the shifter. Note that the ALU status bit N is determined by the ALU output in which bit 14 is the high-order bit, because IR + IR shifts IR left 1 bit. The shifter output does not affect the ALU status bits.

All instructions having 00 in their two high-order bits eventually come to line 4

to have bit 13 tested, with the instructions beginning with 000 going to line 5 and those beginning with 001 going to line 11. Line 5 is an example of a microinstruction with ENC = 0; it just tests TIR but does not change it. Depending on the outcome of this test, the code for LODD or STOD is selected.

For LODD, the microcode must first fetch the word directly addressed by loading the low-order 12 bits of IR into MAR. In this case the high-order 4 bits are all zero but for STOD and other instructions they are not. However, because MAR is only 12 bits wide, the opcode bits do not affect the choice of word read. In line 7, the microprogram has nothing to do, so it just waits. When the word arrives, it is copied to AC and the microprogram jumps to the top of the loop. STOD, ADDD, and SUBD are similar. The only noteworthy point concerning them is how subtraction is done. It uses the fact that

$$x - y = x + (-y) = x + (\bar{y} + 1) = x + 1 + \bar{y}$$

in two's complement. The addition of 1 to AC is done on line 16, which would otherwise be wasted like line 13.

The microcode for JPOS begins on line 21. If AC < 0, the branch fails and the JPOS is terminated immediately by jumping back to the main loop. If, however, AC ≥ 0, the low-order 12 bits of IR are extracted by ANDing them with the 007777 mask and storing the result in PC. It does not cost anything extra to remove the opcode bits here, so we might as well do it. If it had cost an extra microinstruction, however, we would have had to look very carefully to see if having garbage in the high-order 4 bits of PC could cause trouble later.

In a certain sense, JZER (line 23) works the opposite of JPOS. With JPOS, if the condition is met, the jump fails and control returns to the main loop. With JZER, if the condition is met, the jump is taken. Because the code for performing the jump is the same for all the jump instructions, we can save microcode by just going to line 22 whenever feasible. This style of programming would generally be considered uncouth in an application program, but in a microprogram no holds are barred. Performance is everything.

JUMP and LOCO are straightforward, so the next interesting execution routine is for LODL. First, the absolute memory address to be referenced is computed by adding the offset contained in the instruction to SP. Then the memory read is initiated. Because the rest of the code is the same for LODL and LODD, we might as well use lines 7 and 8 for both of them. Not only does this save control store with no loss of execution speed but it also means fewer routines to debug. Analogous code is used for STOL, ADDL, and SUBL. The code for JNEG and JNZE is similar to JZER and JPOS, respectively (not the other way around). CALL first decrements SP, then pushes the return address onto the stack, and finally jumps to the procedure. Line 49 is almost identical to line 22; if it had been exactly the same, we could have eliminated 49 by putting an unconditional jump to 22 in 48. Unfortunately, we must continue to assert WR for another microinstruction.

```
 0: mar := pc; rd;                                {main loop}
 1: pc := pc + 1; rd;                             {increment pc}
 2: ir := mbr; if n then goto 28;                 {save, decode mbr}
 3: tir := lshift(ir + ir); if n then goto 19;
 4: tir := lshift(tir); if n then goto 11;        {000x or 001x?}
 5: alu := tir; if n then goto 9;                 {0000 or 0001?}

 6: mar := ir; rd;                                {0000 = LODD}
 7: rd;
 8: ac := mbr; goto 0;

 9: mar := ir; mbr := ac; wr;                     {0001 = STOD}
10: wr; goto 0;

11: alu := tir; if n then goto 15;                {0010 or 0011?}

12: mar := ir; rd;                                {0010 = ADDD}
13: rd;
14: ac := mbr + ac; goto 0;

15: mar := ir; rd;                                {0011 = SUBD}
16: ac := ac + 1; rd;                             {Note: x − y = x + 1 + not y}
17: a := inv(mbr);
18: ac := ac + a; goto 0;

19: tir := lshift(tir); if n then goto 25;        {010x or 011x?}
20: alu := tir; if n then goto 23;                {0100 or 0101?}

21: alu := ac; if n then goto 0;                  {0100 = JPOS}
22: pc := band(ir, amask); goto 0;                {perform the jump}

23: alu := ac; if z then goto 22;                 {0101 = JZER}
24: goto 0;                                       {jump failed}

25: alu := tir; if n then goto 27;                {0110 or 0111?}
26: pc := band(ir, amask); goto 0;                {0110 = JUMP}

27: ac := band(ir, amask); goto 0;                {0111 = LOCO}

28: tir := lshift(ir + ir); if n then goto 40;    {10xx or 11xx?}
29: tir := lshift(tir); if n then goto 35;        {100x or 101x?}
30: alu := tir; if n then goto 33;                {1000 or 1001?}

31: a := ir + sp;                                 {1000 = LODL}
32: mar := a; rd; goto 7;

33: a := ir + sp;                                 {1001 = STOL}
34: mar := a; mbr := ac; wr; goto 10;

35: alu := tir; if n then goto 38;                {1010 or 1011?}

36: a := ir + sp;                                 {1010 = ADDL}
37: mar := a; rd; goto 13;

38: a := ir + sp;                                 {1011 = SUBL}
39: mar := a; rd; goto 16;
```

Fig. 4-16. The microprogram.

```
40: tir := lshift(tir); if n then goto 46;        {110x or 111x?}
41: alu := tir; if n then goto 44;                {1100 or 1101?}

42: alu := ac; if n then goto 22;                 {1100 = JNEG}
43: goto 0;

44: alu := ac; if z then goto 0;                  {1101 = JNZE}
45: pc := band(ir, amask); goto 0;

46: tir := lshift(tir); if n then goto 50;

47: sp := sp + (−1);                              {1110 = CALL}
48: mar := sp; mbr := pc; wr;
49: pc := band(ir, amask); wr; goto 0;

50: tir := lshift(tir); if n then goto 65;        {1111, examine addr}
51: tir := lshift(tir); if n then goto 59;
52: alu := tir; if n then goto 56;

53: mar := ac; rd;                                {1111000 = PSHI}
54: sp := sp + (−1); rd;
55: mar := sp; wr; goto 10;

56: mar :=sp; sp := sp + 1; rd;                   {1111001 = POPI}
57: rd;
58: mar := ac; wr; goto 10;

59: alu := tir; if n then goto 62;

60: sp := sp + (−1);                              {1111010 = PUSH}
61: mar := sp; mbr := ac; wr; goto 10;

62: mar := sp; sp := sp + 1; rd;                  {1111011 = POP}
63: rd;
64: ac := mbr; goto 0;

65: tir := lshift(tir); if n then goto 73;
66: alu := tir; if n then goto 70;

67: mar := sp; sp := sp + 1; rd;                  {1111100 = RETN}
68: rd;
69: pc := mbr; goto 0;

70: a := ac;                                      {1111101 = SWAP}
71: ac := sp;
72: sp := a; goto 0;

73: alu := tir; if n then goto 76;

74: a := band(ir, smask);                         {1111110 = INSP}
75: sp := sp + a; goto 0;

76: a := band(ir, smask);                         {1111111 = DESP}
77: a := inv(a);
78: a := a + 1; goto 75;
```

Fig. 4-16. (cont.)

The rest of the macroinstructions all have 1111 as the high-order 4 bits, so decoding of the "address bits" is required to tell them apart. The actual execution routines are straightforward so we will not comment on them further.

4.4.3. Remarks about the Microprogram

Although we have discussed the microprogram in considerable detail, a few more points are worth making. In Fig. 4-16 we increment PC in line 1. It could equally well have been done in line 0, thus freeing line 1 for something else while waiting. In this machine there is nothing else to do but in a real machine the microprogram might use this opportunity to check for I/O devices awaiting service, refresh dynamic RAM, or something else.

If we leave line 1 the way it is, however, we can speed up the machine by modifying line 8 to read

mar := *pc*; *ac* := *mbr*; *rd*; **goto** 1;

In other words, we can start fetching the next instruction before we have really finished with the current one. This ability provides a primitive form of instruction pipelining. The same trick can be applied to other execution routines as well.

It is clear that a substantial amount of the execution time of each macroinstruction is devoted to decoding it bit by bit. This observation suggests that it might be useful to be able to load MPC under microprogram control. On many existing computers the microarchitecture has hardware support for extracting macroinstruction opcodes and stuffing them directly into MPC to effect a multiway branch. If, for example, we could shift the IR 9 bits to the right, clear the upper 9 bits, and put the resulting number into MPC, we would have a 128-way branch to locations 0 to 127. Each of these words would contain the first microinstruction for the corresponding macroinstruction. Although this approach wastes control store, it speeds up the machine greatly, so something like it is nearly always used in practice.

We have not said a word about how I/O is implemented. Nor do we have to. By using memory mapping, the CPU is not aware of the difference between true memory addresses and I/O device registers. The microprogram handles reads and writes to the top four words of the address space the same way it handles any other reads and writes.

4.4.4. Perspective

The time seems appropriate to stop for a minute and reflect on what microprogramming is all about. The basic idea is to start out with a simple hardware machine. In our example, it consists of little more than 22 registers, a small ROM for the control store, a glorified adder, an incrementer, a shifter, and some

combinational circuitry for multiplexing, decoding, and sequencing. Using this hardware we were able to construct a software interpreter for carrying out the instructions of a level 2 machine. With the aid of a compiler, we can translate high-level language programs to level 2 instructions and then interpret these instructions one at a time.

Thus to run a program written in a high-level language, we must first translate it to level 2, and then interpret the resulting instructions. Level 2 effectively serves as an interface between the compiler and the interpreter. Although in principle the compiler could generate microcode directly, doing so is complicated and wasteful of space. Each of our macroinstructions occupies one 16-bit word, whereas the corresponding microcode, excluding the instruction decoding logic, requires about four 32-bit microinstructions, on the average. If we were to compile directly to level 1, the total storage needed would increase about eightfold. Furthermore, the increased storage needed is writable control store, which is far more expensive due to its high speed. Using main memory for microcode is not desirable because it results in a slow machine.

In light of these concrete examples, it should be clear why machines are now normally designed as a series of levels. It is done for efficiency and simplicity, because each level deals with another level of abstraction. The level 0 designer worries about how to squeeze the last few nanoseconds out of the ALU by using some spiffy new algorithm to reduce carry-propagation time. The microprogrammer worries about how to get the most mileage out of each microinstruction, typically by exploiting as much of the hardware's inherent parallelism as possible. The macroinstruction set designer worries about how to provide an interface that both the compiler writer and microprogrammer can learn to love, and be efficient at the same time. Clearly, each level has different goals, problems, techniques, and, in general, a different way of looking at the machine. By splitting the total machine design problem into several subproblems, we can attempt to master the inherent complexity in designing a modern computer.

4.5. DESIGN OF THE MICROPROGRAMMING LEVEL

Like just about everything else in computer science, the design of the microarchitecture is full of trade-offs. In the following sections we will look at some of the design issues and the corresponding trade-offs.

4.5.1. Horizontal versus Vertical Microprogramming

Probably the key trade-off is how much encoding to put in the microinstructions. If one were to build the Mic-1 on a single VLSI chip, one could ignore the abstractions such as registers, ALU, and so on, and just look at all the gates. To make the machine run, certain signals are needed, such as the 16 OE signals to gate