Opcode	Instruction	RTN
0000	JnS *X*	MBR ← PC MAR ← *X* M[MAR] ← MBR MBR ← *X* AC ← 1 AC ← AC + MBR PC ← AC
0001	Load *X*	MAR ← *X* MBR ← M[MAR] AC ← MBR
0010	Store *X*	MAR ← *X*, MBR ← AC M[MAR] ← MBR
0011	Add *X*	MAR ← *X* MBR ← M[MAR] AC ← AC + MBR
0100	Subt *X*	MAR ← *X* MBR ← M[MAR] AC ← AC - MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	

1000	Skipcond	If IR[11–10] = 00 then
		  If AC < 0 then PC ← PC + 1
		Else If IR[11–10] = 01 then
		  If AC = 0 then PC ← PC + 1
		Else If IR[11–10] = 10 then
		  If AC > 0 then PC ← PC + 1

1001	Jump *X*	PC ← IR[11–0]
1010	Clear	AC ← 0
1011	AddI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC + MBR
1100	JumpI *X*	MAR ← *X* MBR ← M[MAR] PC ← MBR

TABLE 4.7 MARIE's Full Instruction Set

EXAMPLE 4.2 This example illustrates the use of an `if/else` construct to allow for selection. In particular, it implements the following:

```
if X = Y then
   X = X × 2
else
   Y = Y − X;
```

```
Address          Instruction
100      If,     Load      X       /Load the first value
101              Subt      Y       /Subtract the value of Y, store result in AC
102              Skipcond  400     /If AC = 0, skip the next instruction
103              Jump      Else    /Jump to Else part if AC is not equal to 0
104      Then,   Load      X       /Reload X so it can be doubled
105              Add       X       /Double X
106              Store     X       /Store the new value
107              Jump      Endif   /Skip over the false, or else, part to end of
                                   /if
108      Else,   Load      Y       /Start the else part by loading Y
109              Subt      X       /Subtract X from Y
10A              Store     Y       /Store Y - X in Y
10B      Endif,  Halt              /Terminate program (it doesn't do much!)
10C      X,      Dec       12      /Load the loop control variable
10D      Y,      Dec       20      /Subtract one from the loop control variable
```

Example 4.3 demonstrates how `JnS` and `JumpI` are used to allow for subroutines. This program includes an `END` statement, another example of an assembler directive. This statement tells the assembler where the program ends. Other potential directives include statements to let the assembler know where to find the first program instruction, how to set up memory, and whether blocks of code are procedures.

EXAMPLE 4.3 This example illustrates the use of a simple subroutine to double any number and can be coded.

```
Address          Instruction
100              Load      X       /Load the first number to be doubled
101              Store     Temp    /Use Temp as a parameter to pass value to Subr
102              JnS       Subr    /Store return address, jump to procedure
103              Store     X       /Store first number, doubled
104              Load      Y       /Load the second number to be doubled
105              Store     Temp    /Use Temp as a parameter to pass value to Subr
106              JnS       Subr    /Store return address, jump to procedure
107              Store     Y       /Store second number, doubled
108              Halt              /End program
109      X,      Dec       20
10A      Y,      Dec       48
10B      Temp    Dec       0
```

```
10C       Subr,  Hex       0      /Store return address here
10D              Load      Temp   /Subroutine to double numbers
10E              Add       Temp
10F              JumpI     Subr
                 END
```

Note: Line numbers in program are given for information only and are not required for use in the MarieSim environment.

Using MARIE's simple instruction set, you should be able to implement any high-level programming language construct, such as loop statements and while statements. These are left as exercises at the end of the chapter.

4.13 A DISCUSSION ON DECODING: HARDWIRED VERSUS MICROPROGRAMMED CONTROL

How does the control unit really function? We have done some hand waving and simply assumed everything works as described, with a basic understanding that, for each instruction, the control unit causes the CPU to execute a sequence of steps correctly. In reality, there must be control signals to assert lines on various digital components to make things happen as described (recall the various digital components from Chapter 3). For example, when we perform an `Add` instruction in MARIE in assembly language, we assume the addition takes place because the control signals for the ALU are set to "add" and the result is put into the AC. The ALU has various control lines that determine which operation to perform. The question we need to answer is, "How do these control lines actually become asserted?"

There are two methods by which control lines can be set. The first approach, **hardwired control**, directly connects the control lines to the actual machine instructions. The instructions are divided into fields, and bits in the fields are connected to input lines that drive various digital logic components. The second approach, **microprogrammed control**, employs software consisting of microinstructions that carry out an instruction's microoperations. We look at both of these control methods in more detail after we describe machine control in general.

4.13.1 Machine Control

In Sections 4.8 and 4.12, we provided register transfer language for each of MARIE's instructions. The microoperations described by the register transfer language actually define the operation of the control unit. Each microoperation is associated with a distinctive signal pattern. The signals are fed to combinational circuits within the control unit that carry out the logical operations appropriate to the instruction.

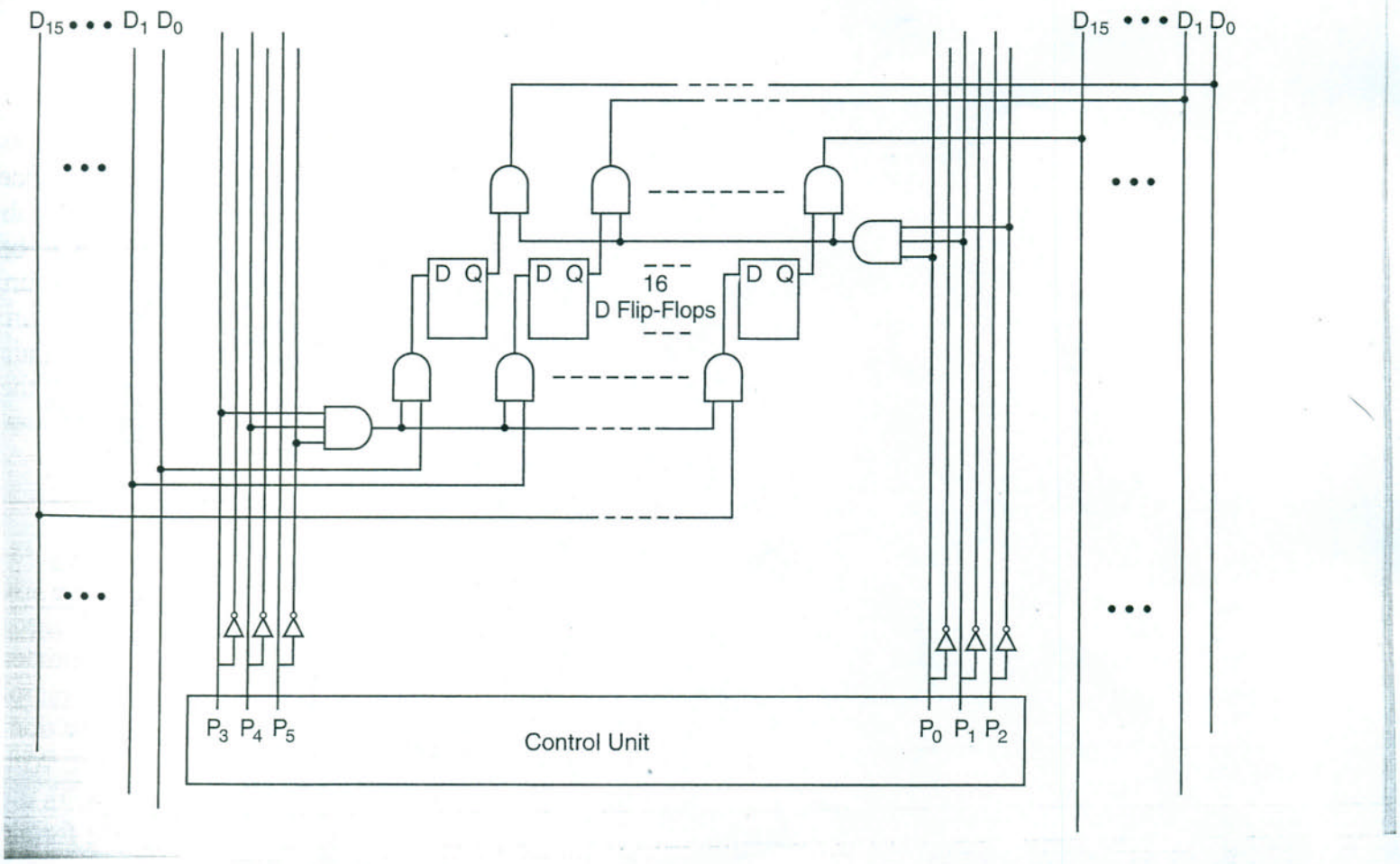
A schematic of MARIE's data path is shown in Figure 4.9. We see that each register and main memory has an address (0 through 7) along the datapath. These addresses, in the form of signal patterns, are used by the control unit to enable the flow of bytes through the system. For the sake of example, we define two sets of signals: P_0, P_1, P_0 that can enable reading from memory or a register and P_3, P_4,

P_3 that can enable writing to a register. The control lines that convey these signals are connected to registers through combinational logic circuits.

A close-up view of the connection of MARIE's MBR (with address 3) to the datapath is shown in Figure 4.15. You can see how this register is enabled for reading when signals P_1 and P_0 are high, and writing to the MBR is enabled when signals P_4 and P_3 are high. (Note that these signals correspond to the binary string of the address of the MBR, 011_2.) No other signal pattern is recognized by this register's circuits. (The combinational logic that enables the other entities on the datapath is left as an exercise.)

If you study MARIE's instruction set, you will see that the ALU has only three operations: add, subtract, and clear. We also need to consider the case where the ALU is not involved in an instruction, so we'll define "do nothing" as a fourth ALU state. Thus, with only four operations, MARIE's ALU can be controlled using only two control signals that we'll call A_0 and A_1.

A computer's clock sequences microoperations by raising the right signals at the right time. MARIE's instructions vary in the number of clock cycles each requires. The activities taking place during each clock cycle are coordinated with signals from a cycle counter. One way of doing this is to connect the clock to a



synchronous counter, and the counter to a decoder. Suppose that the largest number of clock cycles required by any instruction is eight. Then we need a 3-bit counter and a 3 × 8 decoder. The output of the decoder, signals T_0 through T_7, is ANDed with combinational components and registers to produce the behavior required by the instruction. If fewer than eight clock cycles are needed for an instruction, the cycle counter reset signal, C_r, is asserted to get ready for the next machine instruction.

To pull all of this together, consider MARIE's `Add` instruction. The RTN is:

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

After the `Add` instruction is fetched, *X* is in the rightmost 12 bits of the IR and the IR's datapath address is 7, so we need to raise all three datapath read signals, $P_2\ P_1\ P_0$, to place IR bits 0 through 11 on the bus. The MAR, with an address of 1, is activated for writing by raising only P_3. Using the signals as we have just defined them, we can now add the signal patterns to our RTN as follows:

```
P0P1P2P3T0:  MAR ← X
   P3P4T1:   MBR ← M[MAR]
A0P0P1P5T2:  AC ← AC + MBR
     CrT3:   [Reset the clock cycle counter.]
```

All signals, except for data signals (D_0. . .D_{15}), are assumed to be low unless specified in the RTN. Figure 4.16 is a timing diagram that illustrates the sequence of signal patterns just described. As you can see, at clock cycle C_0, all signals except P_0, P_1, P_2, P_3, and T_0 are low. Enabling P_0, P_1, and P_2 allows the IR to be read from, and asserting P_3 allows the MAR to be written to. This action occurs only when T_0 is asserted. At clock cycle C_1, all signals except P_3, P_4, and T_1 are low. This machine state, occurring at time T_1, connects the bytes read from main memory (address zero) to the inputs on the MBR. The last microinstruction of the `Add` sequence occurs at clock cycle T_3, when the timing signals are reset to 0.

4.13.2 Hardwired Control

The sequence of control signals for MARIE's `Add` instruction is identical regardless of whether we employ hardwired or microprogrammed control. If we use hardwired control, the bit pattern in the machine instruction (`Add = 0011`) feeds directly into combinational logic within the control unit. The control unit initiates the sequence of signal events that we just described. A generic hardwired control unit is shown in Figure 4.17. The most interesting part of this diagram is the connection between the instruction decoder and the logic inside the control unit. With timing being key to all activities in the system, the timing signals, along with the bits in the instruction, produce the required behavior. The hardwired logic for the

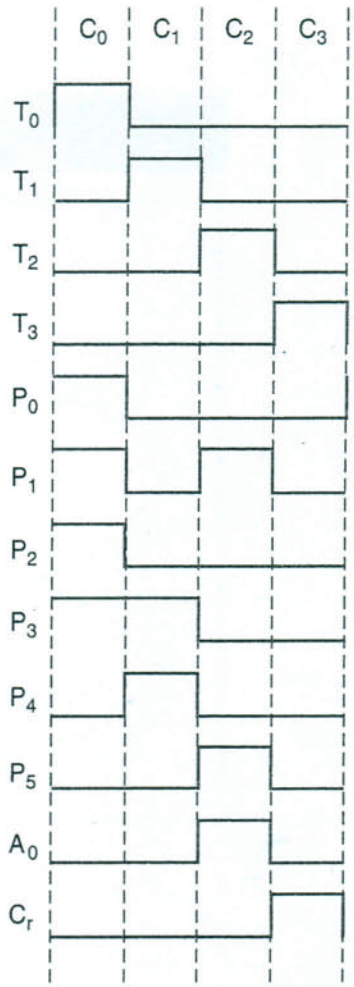


FIGURE 4.16 Timing Diagram for the Microoperations of MARIE's `Add` Instruction

`Add` instruction is shown in Figure 4.18. You can see how each clock cycle is ANDed with the instruction bits to raise the signals as appropriate. With each clock tick, a different group of combinational logic circuits is activated.

The advantage of hardwired control is that it is very fast. The disadvantage is that the instruction set and the control logic are tied together directly by complex circuits that are difficult to design and modify. If someone designs a hardwired computer and later decides to extend the instruction set (as we did with MARIE), the physical components in the computer must be changed. This is prohibitively expensive, because not only must new chips be fabricated, but the old ones must also be located and replaced.

4.13.3 Microprogrammed Control

Signals control the movement of bytes (which are actually signal patterns that we *interpret* as bytes) along the datapath in a computer system. The manner in which

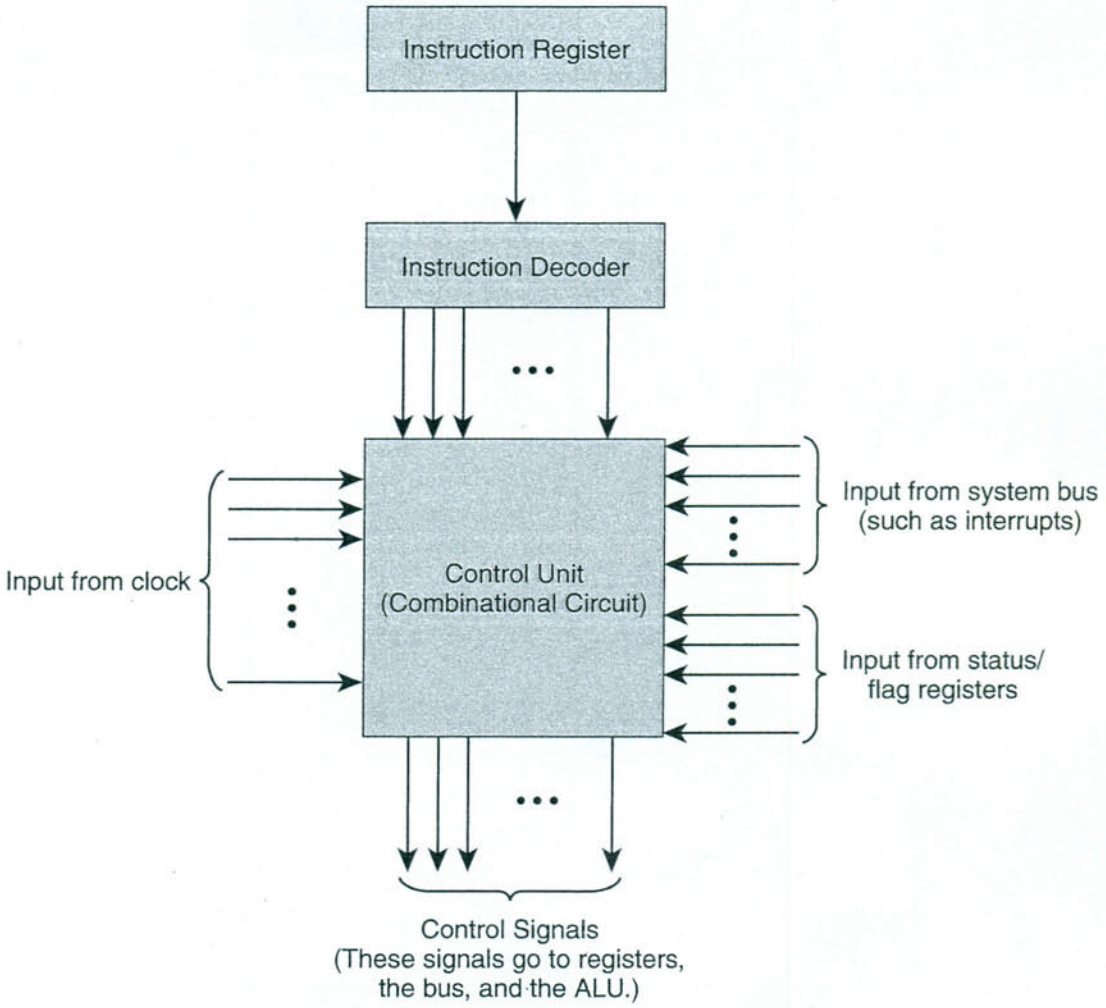


FIGURE 4.17 Hardwired Control Unit

these control signals are produced is what distinguishes hardwired control from microprogrammed control. In hardwired control, timing signals from the clock are ANDed using combinational logic circuits to raise and lower signals. In microprogrammed control, instruction **microcode** produces changes in the data-path signals. A generic block diagram of a microprogrammed control unit is shown in Figure 4.19.

All machine instructions are input into a special program, the **microprogram**, that converts machine instructions of 0s and 1s into control signals. The microprogram is essentially an interpreter, written in microcode, that is stored in **firmware** (ROM, PROM, or EPROM), which is often referred to as the **control store**. A microcode microinstruction is retrieved during each clock cycle. The particular instruction retrieved is a function of the current state of the machine and the value of the **microsequencer**, which is somewhat like a program counter that selects the next instruction from the control store. If MARIE were microprogrammed, the microinstruction format might look like the one shown in