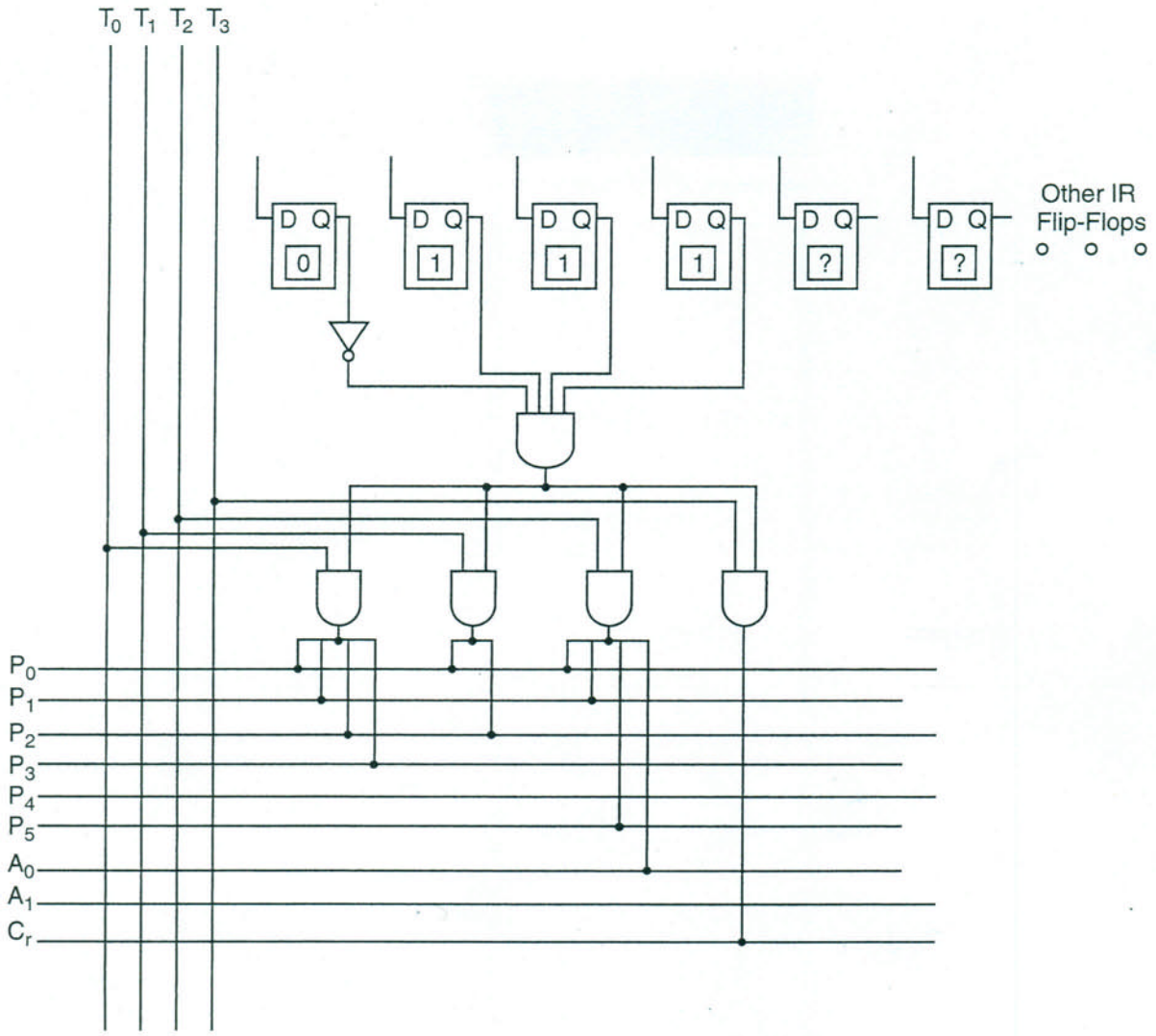

FIGURE 4.18 Combinational Logic for Signal Controls of MARIE's `Add` Instruction

Figure 4.20.

`MicroOp1` and `MicroOp2` are binary codes for each unique microoperation specified in the RTN for MARIE's instruction set. A comprehensive list of this RTN (as given in Table 4.7) along with the RTN for the fetch–decode–execute cycle reveals that there are only 22 unique microoperations required to implement MARIE's entire instruction set. Two additional microoperations are also necessary. One of these codes, `NOP`, indicates "no operation." `NOP` is useful when the system must wait for a set of signals to stabilize, when waiting for a value to be fetched from memory, or when we need a placeholder. Second, and most important, we need a microoperation that compares the bit pattern in the first 4 bits of the instruction register (IR[15-12]) to a literal value that is in the first 4 bits of the

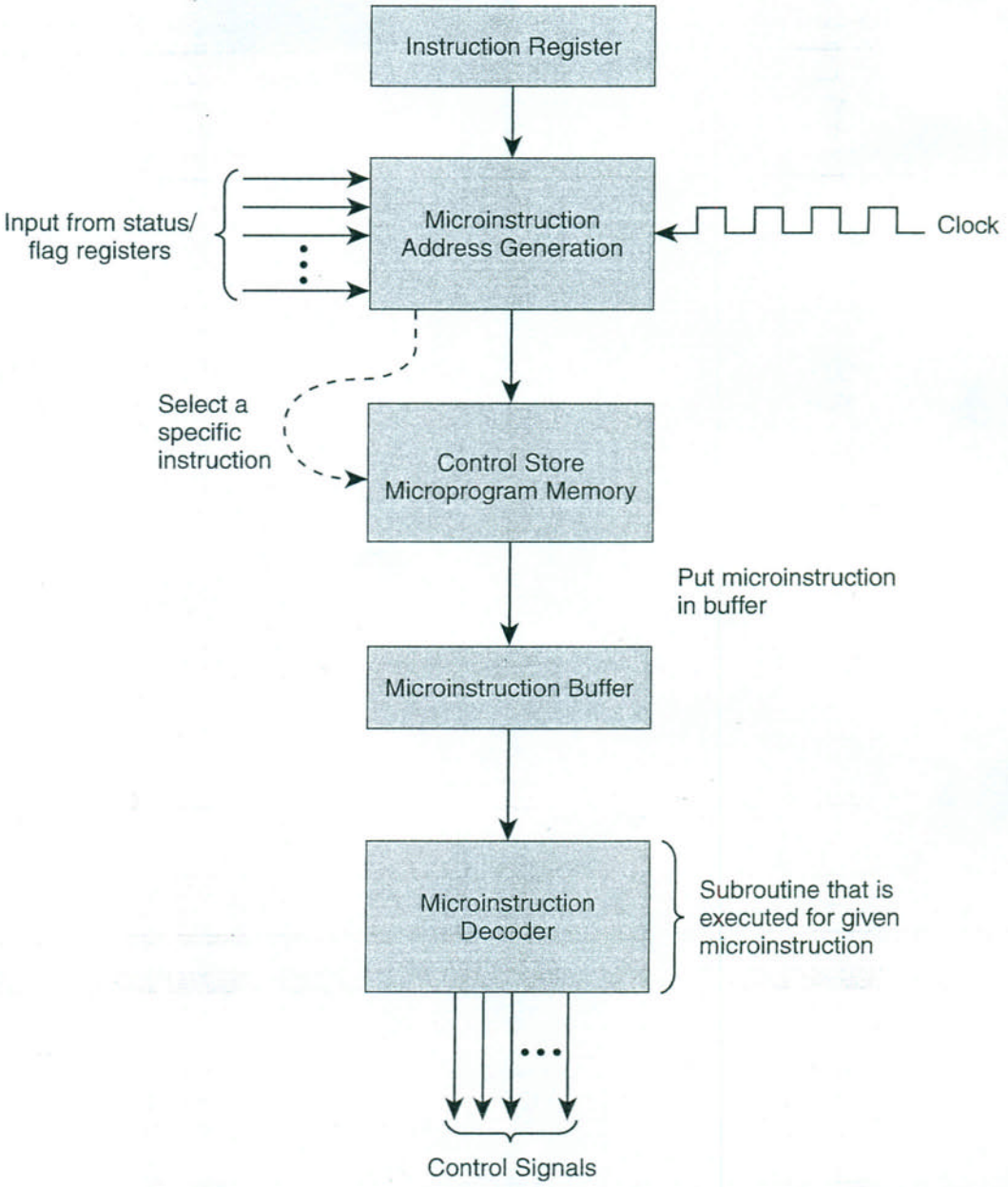

FIGURE 4.19 Microprogrammed Control Unit

Field Name	MicroOp 1	MicroOp 2	Jump	Dest
Meaning	First Microoperation	Second Microoperation	Boolean: Set to indicate a jump.	Destination address for jump.
Bit	17 13	12 8	7	6 0

FIGURE 4.20 MARIE's Microinstruction Format

MicroOp2 field. This instruction is crucial to the execution control of MARIE's microprogram. Each of MARIE's microoperations is assigned a binary code, as shown in Table 4.8.

MARIE's entire microprogram consists of fewer than 128 statements, so each statement can be uniquely identified by seven bits. This means that each microinstruction has a seven-bit address. When the Jump bit is set, it indicates that the Dest field contains a valid address. This address is then moved to the **microsequencer**, which is the program counter that controls the flow of execution in the microprogram. Control then branches to the address found in the Dest field.

MARIE's control store memory holds the entire microprogram in contiguous space. This program consists of a jump table and blocks of code that correspond to each of MARIE's operations. The first nine statements (in RTL form) of MARIE's microprogram are given in Figure 4.21 (we have used the RTL for clarity; the microprogram is actually stored in binary). When MARIE is booted up, hardware sets the microsequencer to point to address 0000000 of the microprogram. Execution commences from this entry point. We see that the first four statements of the microprogram are the first four statements of the fetch–decode–execute cycle. The statements starting at address 0000100 are the jump table containing the addresses of the statements that carry out the machine instructions. They effectively decode the instruction by branching to the code block that sets the control signals to carry out the machine instruction.

At line number 0000111, the statement If IR[15-12] = MicroOp2[4-1] compares the value in the leftmost 4 bits of the second microoperation field with the value in the opcode field of the instruction that was fetched in the first three lines of the microprogram. In this particular statement, we are comparing the

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01100	MBR ← M[MAR]
00001	AC ← 0	01101	OutREG ← AC
00010	AC ← AC − MBR	01110	PC ← IR[11–0]
00011	AC ← AC + MBR	01111	PC ← MBR
00100	AC ← InREG	10000	PC ← PC + 1
00101	IR ← M[MAR]	10001	If AC = 00
00110	M[MAR] ← MBR	10010	If AC > 0
00111	MAR ← IR[11–0]	10011	If AC < 0
01000	MAR ← MBR	10100	If IR[11–10] = 00
01001	MAR ← PC	10101	If IR[11–10] = 01
01010	MAR ← X	10110	If IR[11–10] = 10
01011	MBR ← AC	10111	If IR[15–12] = MicropOp2[4-1]

TABLE 4.8 Microoperation Codes and Corresponding MARIE RTL

Address	MicroOp 1	MicroOp 2	Jump	Dest
0000000	MAR ← PC	NOP	0	0000000
0000001	IR ← M[MAR]	NOP	0	0000000
0000010	PC ← PC + 1	NOP	0	0000000
0000011	MAR ←IR[11-0]	NOP	0	0000000
0000100	If IR[15-12] = MicroOP2[4-1]	00000	1	0100000
0000101	If IR[15-12] = MicroOP2[4-1]	00010	1	0100111
0000110	If IR[15-12] = MicroOP2[4-1]	00100	1	0101010
0000111	If IR[15-12] = MicroOP2[4-1]	00110	1	0101100
0001000	If IR[15-12] = MicroOP2[4-1]	01000	1	0101111
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
0101010	MAR ←X	MBR ← AC	0	0000000
0101011	M[MAR]← MBR	NOP	1	0000000
0101100	MAR ←X	NOP	0	0000000
0101101	MBR ←M[MAR]	NOP	0	0000000
0101110	AC ←AC + MBR	NOP	1	0000000
0101111	MAR ←MAR	NOP	0	0000000
. . .	. . .	. . .	. . .	. . .

FIGURE 4.21 Selected Statements in MARIE's Microprogram

opcode against MARIE's binary code for the `Add` operation, 0011. If we have a match, the `Jump` bit is set to true and control branches to address 0101100.

At address 0101100, we see the microoperations (RTN) for the `Add` instruction. As these microoperations are executed, control lines are set exactly as described in Section 4.13.1. The last instruction, at 0101011, has the `Jump` bit set once again. The setting of this bit causes the value of all 0s (the jump `Dest`) to be moved to the microsequencer. This effectively branches back to the start of the fetch cycle at the top of the program.

We must emphasize that a microprogrammed control unit works like a system in miniature. To fetch an instruction from the control store, a certain set of signals must be raised. The microsequencer points at the instruction to retrieve and is subsequently incremented. This is why microprogrammed control tends to be slower than hardwired control—all instructions must go through an additional level of interpretation. But performance is not everything. Microprogramming is flexible, simple in design, and lends itself to very powerful instruction sets. The great advantage of microprogrammed control is that if the instruction set requires

modification, only the microprogram needs to be updated to match: No changes to the hardware are required. Thus, microprogrammed control units are less costly to manufacture and maintain. Because cost is critical in consumer products, microprogrammed control dominates the personal computer market.

4.14 REAL-WORLD EXAMPLES OF COMPUTER ARCHITECTURES

The MARIE architecture is designed to be as simple as possible so that the essential concepts of computer architecture would be easy to understand without being completely overwhelming. Although MARIE's architecture and assembly language are powerful enough to solve any problems that could be carried out on a modern architecture using a high-level language such as C++, Ada, or Java, you probably wouldn't be very happy with the inefficiency of the architecture or with how difficult the program would be to write and to debug! MARIE's performance could be significantly improved if more storage were incorporated into the CPU by adding more registers. Making things easier for the programmer is a different matter. For example, suppose a MARIE programmer wants to use procedures with parameters. Although MARIE allows for subroutines (programs can branch to various sections of code, execute the code, and then return), MARIE has no mechanism to support the passing of parameters. Programs can be written without parameters, but we know that using them not only makes the program more efficient (particularly in the area of reuse), but also makes the program easier to write and debug.

To allow for parameters, MARIE would need a **stack**, a data structure that maintains a list of items that can be accessed from only one end. A pile of plates in your kitchen cabinet is analogous to a stack: You put plates on the top and you take plates off the top (normally). For this reason, stacks are often called **last-in-first-out** structures. (Please see Appendix A at the end of this book for a brief overview of the various data structures.)

We can emulate a stack using certain portions of main memory if we restrict the way data is accessed. For example, if we assume memory locations 0000 through 00FF are used as a stack, and we treat 0000 as the top, then **pushing** (adding) onto the stack must be done from the top, and **popping** (removing) from the stack must be done from the top. If we push the value 2 onto the stack, it would be placed at location 0000. If we then push the value 6, it would be placed at location 0001. If we then performed a pop operation, the 6 would be removed. A **stack pointer** keeps track of the location to which items should be pushed or popped.

MARIE shares many features with modern architectures, but is not an accurate depiction of them. In the next two sections, we introduce two contemporary computer architectures to better illustrate the features of modern architectures that, in an attempt to follow Leonardo da Vinci's advice, were excluded from MARIE. We begin with the Intel architecture (the x86 and the Pentium families) and then follow

with the MIPS architecture. We chose these architectures because, although they are similar in some respects, they are built on fundamentally different philosophies. Each member of the x86 family of Intel architectures is known as a **CISC** (**complex instruction set computer**) machine, whereas the Pentium family and the MIPS architectures are examples of **RISC** (**reduced instruction set computer**) machines.

CISC machines have a large number of instructions, of variable length, with complex layouts. Many of these instructions are quite complicated, performing multiple operations when a single instruction is executed (e.g., it is possible to do loops using a *single* assembly language instruction). The basic problem with CISC machines is that a small subset of complex CISC instructions slows the systems down considerably. Designers decided to return to a less complicated architecture and to hardwire a small (but complete) instruction set that would execute extremely quickly. This meant it would be the compiler's responsibility to produce efficient code for the ISA. Machines utilizing this philosophy are called RISC machines.

RISC is something of a misnomer. It is true that the number of instructions is reduced. However, the main objective of RISC machines is to simplify instructions so they can execute more quickly. Each instruction performs only one operation, they are all the same size, they have only a few different layouts, and all arithmetic operations must be performed between registers (data in memory cannot be used as operands). Virtually all new instruction sets (for any architectures) since 1982 have been RISC, or some sort of combination of CISC and RISC. We cover CISC and RISC in detail in Chapter 9.

4.14.1 Intel Architectures

The Intel Corporation has produced many different architectures, some of which may be familiar to you. Intel's first popular chip, the **8086**, was introduced in 1979 and used in the IBM PC. It handled 16-bit data and worked with 20-bit addresses; thus it could address a million bytes of memory. (A close cousin of the 8086, the 8-bit 8088, was used in many personal computers to lower the cost.) The 8086 CPU was split into two parts: the **execution unit**, which included the general registers and the ALU, and the **bus interface unit**, which included the instruction queue, the segment registers, and the instruction pointer.

The 8086 had four 16-bit general purpose registers named AX (the primary accumulator), BX (the base register used to extend addressing), CX (the count register), and DX (the data register). Each of these registers was divided into two pieces: the most significant half was designated the "high" half (denoted by AH, BH, CH, and DH), and the least significant was designated the "low" half (denoted by AL, BL, CL, and DL). Various 8086 instructions required the use of a specific register, but the registers could be used for other purposes as well. The 8086 also had three pointer registers: the stack pointer (SP), which was used as an offset into the stack; the base pointer (BP), which was used to reference parameters pushed

onto the stack; and the instruction pointer (IP), which held the address of the next instruction (similar to MARIE's PC). There were also two index registers: the SI (source index) register, used as a source pointer for string operations, and the DI (destination index) register, used as a destination pointer for string operations. The 8086 also had a **status flags register**. Individual bits in this register indicated various conditions, such as overflow, parity, carry interrupt, and so on.

An 8086 assembly language program was divided into different **segments**, special blocks or areas to hold specific types of information. There was a code segment (for holding the program), a data segment (for holding the program's data), and a stack segment (for holding the program's stack). To access information in any of these segments, it was necessary to specify that item's offset from the beginning of the corresponding segment. Therefore, segment pointers were necessary to store the addresses of the segments. These registers included the code segment (CS) register, the data segment (DS) register, and the stack segment (SS) register. There was also a fourth segment register, called the extra segment (ES) register, which was used by some string operations to handle memory addressing. Addresses were specified using segment/offset addressing in the form: *xxx:yyy*, where *xxx* was the value in the segment register and *yyy* was the offset.

In 1980, Intel introduced the 8087, which added floating-point instructions to the 8086 machine set as well as an 80-bit wide stack. Many new chips were introduced that used essentially the same ISA as the 8086, including the 80286 in 1982 (which could address 16 million bytes) and the 80386 in 1985 (which could address up to 4 billion bytes of memory). The 80386 was a 32-bit chip, the first in a family of chips often called IA-32 (for Intel Architecture, 32-bit). When Intel moved from the 16-bit 80286 to the 32-bit 80386, designers wanted these architectures to be **backward compatible**, which means that programs written for a less powerful and older processor should run on the newer, faster processors. For example, programs that ran on the 80286 should also run on the 80386. Therefore, Intel kept the same basic architecture and register sets. (New features were added to each successive model, so forward compatibility was not guaranteed.)

The naming convention used in the 80386 for the registers, which had gone from 16 to 32 bits, was to include an "E" prefix (which stood for "extended"). So instead of AX, BX, CX, and DX, the registers became EAX, EBX, ECX, and EDX. This same convention was used for all other registers. However, the programmer could still access the original registers, AX, AL, and AH, for example, using the original names. Figure 4.22 illustrates how this worked, using the AX register as an example.

The 80386 and 80486 were both 32-bit machines, with 32-bit data buses. The 80486 added a high-speed **cache** memory (see Chapter 6 for more details on cache and memory), which improved performance significantly.

The **Pentium** series (see sidebar "What's in a Name?" to find out why Intel stopped using numbers and switched to the name "Pentium") started with the Pentium processor, which had 32-bit registers and a 64-bit data bus and employed