> *"When you wish to produce a result by means of an instrument, do not allow yourself to complicate it."*
>
> *—Leonardo da Vinci*

CHAPTER 4 MARIE: An Introduction to a Simple Computer

4.1 INTRODUCTION

Designing a computer nowadays is a job for a computer engineer with plenty of training. It is impossible in an introductory textbook such as this (and in an introductory course in computer organization and architecture) to present everything necessary to design and build a working computer such as those we can buy today. However, in this chapter, we first look at a very simple computer called MARIE: a **M**achine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy. We then provide brief overviews of Intel and MIPs machines, two popular architectures reflecting the CISC and RISC design philosophies. The objective of this chapter is to give you an understanding of how a computer functions. We have, therefore, kept the architecture as uncomplicated as possible, following the advice in the opening quote by Leonardo da Vinci.

4.2 CPU BASICS AND ORGANIZATION

From our studies in Chapter 2 (data representation) we know that a computer must manipulate binary-coded data. We also know from Chapter 3 that memory is used to store both data and program instructions (also in binary). Somehow, the program must be executed and the data must be processed correctly. The **central processing unit** (**CPU**) is responsible for fetching program instructions, decoding each instruction that is fetched, and performing the indicated sequence of operations on the correct data. To understand how computers work, you must first become familiar with their various components and the interaction among these components. To introduce the simple architecture in the next section, we first examine, in general, the microarchitecture that exists at the control level of modern computers.

All computers have a CPU that can be divided into two pieces. The first is the **datapath**, which is a network of storage units (registers) and arithmetic and logic units (for performing various operations on data) connected by buses (capable of moving data from place to place) where the timing is controlled by clocks. The second CPU component is the **control unit**, a module responsible for sequencing operations and making sure the correct data are where they need to be at the correct time. Together, these components perform the tasks of the CPU: fetching instructions, decoding them, and finally performing the indicated sequence of operations. The performance of a machine is directly affected by the design of the datapath and the control unit. Therefore, we cover these components of the CPU in detail in the following sections.

4.2.1 The Registers

Registers are used in computer systems as places to store a wide variety of data, such as addresses, program counters, or data necessary for program execution. Put simply, a **register** is a hardware device that stores binary data. Registers are located on the processor so information can be accessed very quickly. We saw in Chapter 3 that D flip-flops can be used to implement registers. One D flip-flop is equivalent to a 1-bit register, so a collection of D flip-flops is necessary to store multi-bit values. For example, to build a 16-bit register, we need to connect 16 D flip-flops together. We saw in our binary counter figure from Chapter 3 that these collections of flip-flops must be clocked to work in unison. At each pulse of the clock, input enters the register and cannot be changed (and thus is stored) until the clock pulses again.

Data processing on a computer is usually done on fixed-size binary words stored in registers. Therefore, most computers have registers of a certain size. Common sizes include 16, 32, and 64 bits. The number of registers in a machine varies from architecture to architecture, but is typically a power of 2, with 16 and 32 being most common. Registers contain data, addresses, or control information. Some registers are specified as "special purpose" and may contain only data, only addresses, or only control information. Other registers are more generic and may hold data, addresses, and control information at various times.

Information is written to registers, read from registers, and transferred from register to register. Registers are not addressed in the same way memory is addressed (recall that each memory word has a unique binary address beginning with location 0). Registers are addressed and manipulated by the control unit itself.

In modern computer systems, there are many types of specialized registers: registers to store information, registers to shift values, registers to compare values, and registers that count. There are "scratchpad" registers that store temporary values, index registers to control program looping, stack pointer registers to manage stacks of information for processes, status (or flag) registers to hold the status or mode of operation (such as overflow, carry, or zero conditions), and general purpose registers that are the registers available to the programmer. Most computers have register sets, and each set is used in a specific way. For example, the Pentium architecture has a data register set and an address register set. Certain architectures have very large sets of registers that can be used in quite novel ways to speed up execution of instructions. (We discuss this topic when we cover advanced architectures in Chapter 9.)

4.2.2 The ALU

The **arithmetic logic unit** (**ALU**) carries out the logic operations (such as comparisons) and arithmetic operations (such as add or multiply) required during the program execution. You saw an example of a simple ALU in Chapter 3. Generally an ALU has two data inputs and one data output. Operations performed in the ALU often affect bits in the **status register** (bits are set to indicate actions such as whether an overflow has occurred). The ALU knows which operations to perform because it is controlled by signals from the control unit.

4.2.3 The Control Unit

The **control unit** is the "policeman" or "traffic manager" of the CPU. It monitors the execution of all instructions and the transfer of all information. The control unit extracts instructions from memory, decodes these instructions, making sure data are in the right place at the right time, tells the ALU which registers to use, services interrupts, and turns on the correct circuitry in the ALU for the execution of the desired operation. The control unit uses a **program counter** register to find the next instruction for execution and a status register to keep track of overflows, carries, borrows, and the like. Section 4.13 covers the control unit in more detail.

4.3 THE BUS

The CPU communicates with the other components via a bus. A **bus** is a set of wires that acts as a shared but common datapath to connect multiple subsystems within the system. It consists of multiple lines, allowing the parallel movement of bits. Buses are low cost but very versatile, and they make it easy to connect new devices to each other and to the system. At any one time, only one device (be it a register, the ALU, memory, or some other component) may use the bus. However, this sharing often results in a communications bottleneck. The speed of the bus is affected by its length as well as by the number of devices sharing it. Quite often, devices are divided into **master** and **slave** categories; a master device is one that initiates actions and a slave is one that responds to requests by a master.

A bus can be **point-to-point**, connecting two specific components (as seen in Figure 4.1a) or it can be a **common pathway** that connects a number of devices,

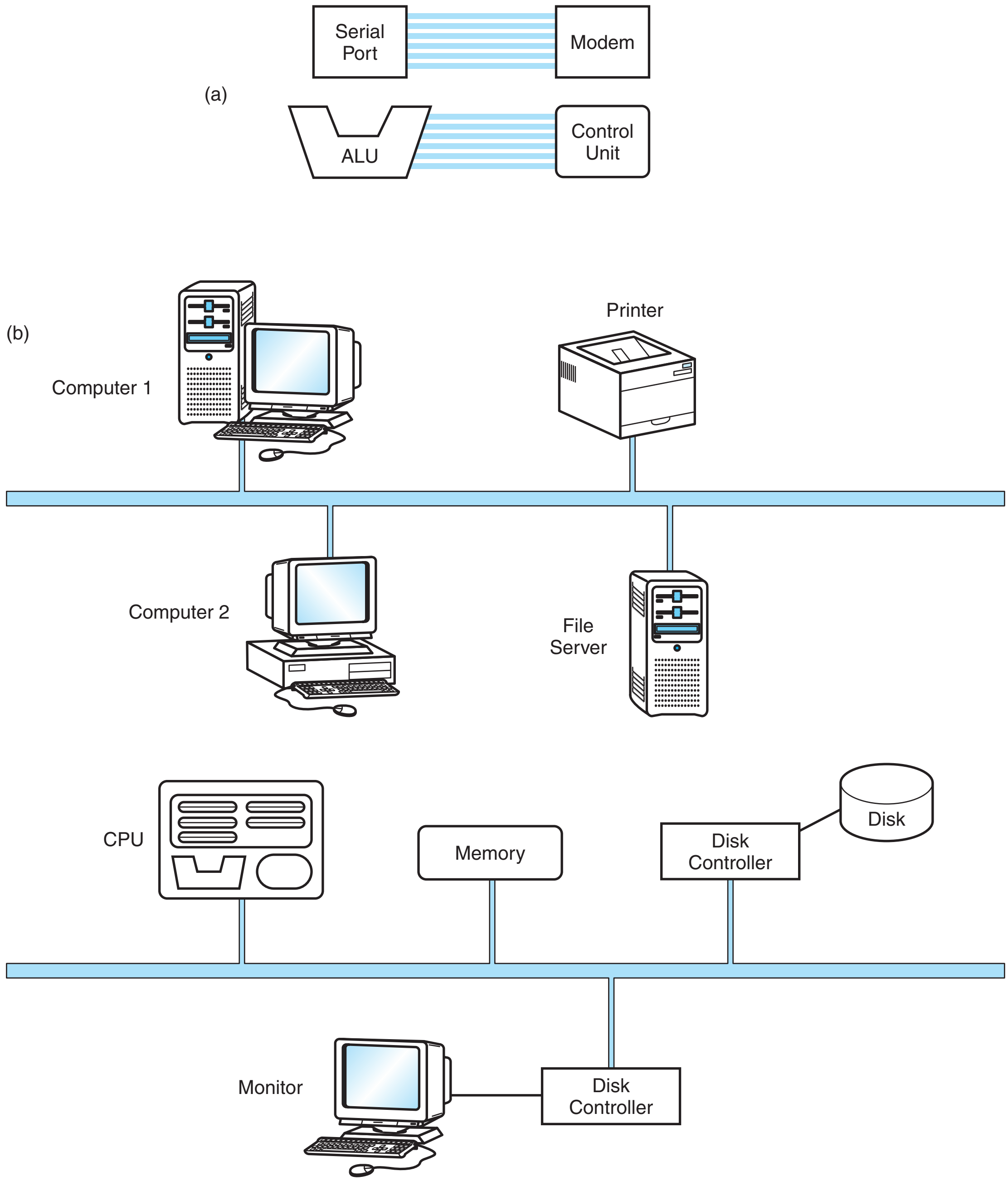


FIGURE 4.1 a) Point-to-Point Buses
b) Multipoint Buses

requiring these devices to share the bus (referred to as a **multipoint** bus and shown in Figure 4.1b).

Because of this sharing, the **bus protocol** (set of usage rules) is very important. Figure 4.2 shows a typical bus consisting of data lines, address lines, control lines, and power lines. Often the lines of a bus dedicated to moving data are called the **data bus**. These data lines contain the actual information that must be moved from one location to another. **Control lines** indicate which device has permission to use the bus and for what purpose (reading or writing from memory or from an input/output [I/O] device, for example). Control lines also transfer acknowledgments for bus requests, interrupts, and clock synchronization signals. **Address lines** indicate the location (e.g., in memory) that the data should be either read from or written to. The **power lines** provide the electrical power necessary. Typical bus transactions include sending an address (for a read or write), transferring data from memory to a register (a memory read), and transferring data to the memory from a register (a memory write). In addition, buses are used for I/O reads and writes from peripheral devices. Each type of transfer occurs within a **bus cycle**, the time between two ticks of the bus clock.

Because of the different types of information buses transport and the various devices that use the buses, buses themselves have been divided into different types. **Processor-memory buses** are short, high-speed buses that are closely matched to the memory system on the machine to maximize the bandwidth (transfer of data) and are usually design specific. **I/O buses** are typically longer than processor-memory buses and allow for many types of devices with varying bandwidths. These buses are compatible with many different architectures. A **backplane bus** (Figure 4.3) is actually built into the chassis of the machine and

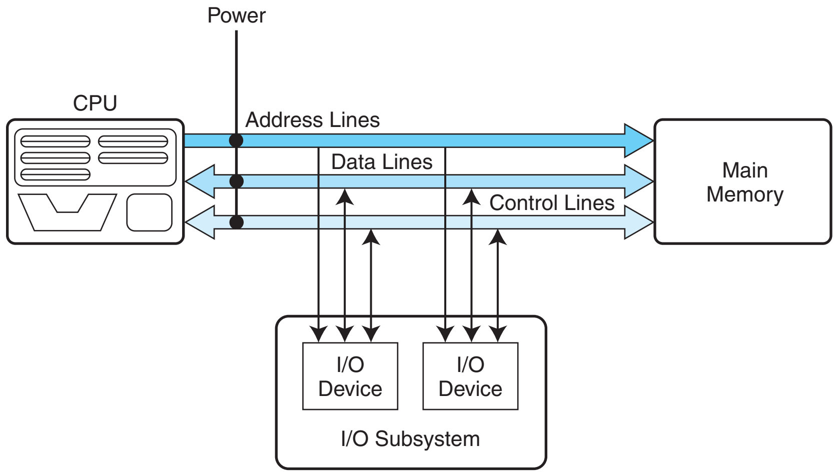


FIGURE 4.2 The Components of a Typical Bus

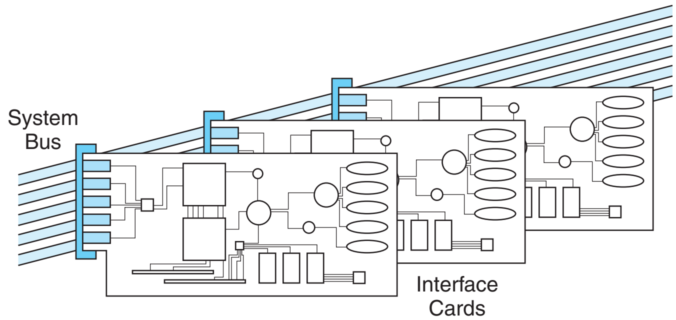


FIGURE 4.3 Backplane Bus

connects the processor, the I/O devices, and the memory (so all devices share one bus). Many computers have a hierarchy of buses, so it is not uncommon to have two buses (e.g., a processor-memory bus and an I/O bus) or more in the same system. High-performance systems often use all three types of buses.

Personal computers have their own terminology when it comes to buses. They have an internal bus (called the **system bus**) that connects the CPU, memory, and all other internal components. External buses (sometimes referred to as **expansion buses**) connect external devices, peripherals, expansion slots, and I/O ports to the rest of the computer. Most PCs also have **local buses**, data buses that connect a peripheral device directly to the CPU. These high-speed buses can be used to connect only a limited number of similar devices. Expansion buses are slower but allow for more generic connectivity. Chapter 7 deals with these topics in great detail.

Buses are physically little more than bunches of wires, but they have specific standards for connectors, timing, and signaling specifications and exact protocols for use. **Synchronous buses** are clocked, and things happen only at the clock ticks (a sequence of events is controlled by the clock). Every device is synchronized by the rate at which the clock ticks, or the **clock rate**. The bus cycle time mentioned is the reciprocal of the bus clock rate. For example, if the bus clock rate is 133 MHz, then the length of the bus cycle is 1/133,000,000 or 7.52 nanoseconds (ns). Because the clock controls the transactions, any **clock skew** (drift in the clock) has the potential to cause problems, implying that the bus must be kept as short as possible so the clock drift cannot get overly large. In addition, the bus cycle time must not be shorter than the length of time it takes information to traverse the bus. The length of the bus, therefore, imposes restrictions on both the bus clock rate and the bus cycle time.

With **asynchronous buses**, control lines coordinate the operations, and a complex **handshaking protocol** must be used to enforce timing. To read a word of data from memory, for example, the protocol would require steps similar to the following:

1. `ReqREAD`: This bus control line is activated and the data memory address is put on the appropriate bus lines at the same time.

2. `ReadyDATA`: This control line is asserted when the memory system has put the required data on the data lines for the bus.
3. `ACK`: This control line is used to indicate that the `ReqREAD` or the `ReadyDATA` has been acknowledged.

Using a protocol instead of the clock to coordinate transactions means that asynchronous buses scale better with technology and can support a wider variety of devices.

To use a bus, a device must reserve it, because only one device can use the bus at a time. As mentioned, bus masters are devices that are allowed to initiate transfer of information (control bus), and bus slaves are modules that are activated by a master and respond to requests to read and write data (so only masters can reserve the bus). Both follow a communications protocol to use the bus, working within very specific timing requirements. In a very simple system (such as the one we present in the next section), the processor is the only device allowed to become a bus master. This is good in terms of avoiding chaos, but bad because the processor now is involved in every transaction that uses the bus.

In systems with more than one master device, **bus arbitration** is required. Bus arbitration schemes must provide priority to certain master devices and, at the same time, make sure lower priority devices are not starved out. Bus arbitration schemes fall into four categories:

1. **Daisy chain arbitration:** This scheme uses a "grant bus" control line that is passed down the bus from the highest priority device to the lowest priority device. (Fairness is not ensured, and it is possible that low-priority devices are "starved out" and never allowed to use the bus.) This scheme is simple but not fair.
2. **Centralized parallel arbitration:** Each device has a request control line to the bus and a centralized arbiter selects who gets the bus. Bottlenecks can result using this type of arbitration.
3. **Distributed arbitration using self-selection:** This scheme is similar to centralized arbitration but instead of a central authority selecting who gets the bus, the devices themselves determine who has highest priority and who should get the bus.
4. **Distributed arbitration using collision detection:** Each device is allowed to make a request for the bus. If the bus detects any collisions (multiple simultaneous requests), the device must make another request. (Ethernet uses this type of arbitration.)

Chapter 7 contains more detailed information on buses and their protocols.

4.4 CLOCKS

Every computer contains an internal clock that regulates how quickly instructions can be executed. The clock also synchronizes all of the components in the system. As the clock ticks, it sets the pace for everything that happens in the system, much like a metronome or a symphony conductor. The CPU uses this clock to regulate its progress, checking the otherwise unpredictable speed of the digital logic gates. The CPU requires a fixed number of clock ticks to execute each instruction. Therefore, instruction performance is often measured in **clock cycles**—the time between clock ticks—instead of seconds. The **clock frequency** (sometimes called the clock rate or clock speed) is measured in megahertz (MHz) or gigahertz (GHz), as we saw in Chapter 1. The **clock cycle time** (or clock period) is simply the reciprocal of the clock frequency. For example, an 800 MHz machine has a clock cycle time of 1/800,000,000 or 1.25ns. If a machine has a 2ns cycle time, then it is a 500 MHz machine.

Most machines are synchronous: there is a master clock signal, which ticks (changing from 0 to 1 to 0 and so on) at regular intervals. Registers must wait for the clock to tick before new data can be loaded. It seems reasonable to assume that if we speed up the clock, the machine will run faster. However, there are limits on how short we can make the clock cycles. When the clock ticks and new data are loaded into the registers, the register outputs are likely to change. These changed output values must propagate through all the circuits in the machine until they reach the input of the next set of registers, where they are stored. The clock cycle must be long enough to allow these changes to reach the next set of registers. If the clock cycle is too short, we could end up with some values not reaching the registers. This would result in an inconsistent state in our machine, which is definitely something we must avoid. Therefore, the minimum clock cycle time must be at least as great as the maximum propagation delay of the circuit, from each set of register outputs to register inputs. What if we "shorten" the distance between registers to shorten the propagation delay? We could do this by adding registers between the output registers and the corresponding input registers. But recall that registers cannot change values until the clock ticks, so we have, in effect, increased the number of clock cycles. For example, an instruction that would require two clock cycles might now require three or four (or more, depending on where we locate the additional registers).

Most machine instructions require one or two clock cycles, but some can take 35 or more. We present the following formula to relate seconds to cycles:

$$\text{CPU time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{average cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

It is important to note that the architecture of a machine has a large effect on its performance. Two machines with the same clock speed do not necessarily execute instructions in the same number of cycles. For example, a multiply operation on an older Intel 286 machine required 20 clock cycles, but on a new Pentium, a multiply operation can be done in 1 clock cycle, which implies the newer machine would be 20 times faster than the 286, even if they both had the same internal system clock. In general, multiplication requires more time than addition, floating-point operations require more cycles than integer ones, and accessing memory takes longer than accessing registers.

Generally, when we mention the **clock**, we are referring to the **system clock**, or the master clock that regulates the CPU and other components. However, certain buses also have their own clocks. **Bus clocks** are usually slower than CPU clocks, causing bottleneck problems.

System components have defined performance bounds, indicating the maximum time required for the components to perform their functions. Manufacturers guarantee their components will run within these bounds in the most extreme circumstances. When we connect all of the components together serially, where one component must complete its task before another can function properly, it is important to be aware of these performance bounds so we are able to synchronize the components properly. However, many people push the bounds of certain system components in an attempt to improve system performance. **Overclocking** is one method people use to achieve this goal.

Although many components are potential candidates, one of the most popular components for overclocking is the CPU. The basic idea is to run the CPU at clock and/or bus speeds above the upper bound specified by the manufacturer. Although this can increase system performance, one must be careful not to create system timing faults or, worse yet, overheat the CPU. The system bus can also be overclocked, which results in overclocking the various components that communicate via the bus. Overclocking the system bus can provide considerable performance improvements, but can also damage the components that use the bus or cause them to perform unreliably.

4.5 THE INPUT/OUTPUT SUBSYSTEM

Input and output *(***I/O***)* **devices** allow us to communicate with the computer system. I/O is the transfer of data between primary memory and various I/O peripherals. Input devices such as keyboards, mice, card readers, scanners, voice recognition systems, and touch screens allow us to enter data into the computer. Output devices such as monitors, printers, plotters, and speakers allow us to get information from the computer.

These devices are not connected directly to the CPU. Instead, there is an **interface** that handles the data transfers. This interface converts the system bus signals to and from a format that is acceptable to the given device. The CPU communicates to these external devices via I/O registers. This exchange of data is performed in two ways. In **memory-mapped I/O**, the registers in the interface appear in the computer's memory map and there is no real difference between accessing memory and accessing an I/O device. Clearly, this is advantageous from the perspective of speed, but it uses up memory space in the system. With **instruction-based I/O**, the CPU has specialized instructions that perform the input and output. Although this does not use memory space, it requires specific I/O instructions, which implies it can be used only by CPUs that can execute these specific instructions. Interrupts play a very important part in I/O, because they are an efficient way to notify the CPU that input or output is available for use. We explore thse I/O methods in detail in Chapter 7.

4.6 MEMORY ORGANIZATION AND ADDRESSING

We saw an example of a rather small memory in Chapter 3. However, we have not yet discussed in detail how memory is laid out and how it is addressed. It is important that you have a good understanding of these concepts before we continue.

You can envision memory as a matrix of bits. Each row, implemented by a register, has a length typically equivalent to the addressable unit size of the machine. Each register (more commonly referred to as a **memory location**) has a unique address; memory addresses usually start at zero and progress upward. Figure 4.4 illustrates this concept.

An address is typically represented by an unsigned integer. Recall from Chapter 2 that four bits are a nibble and eight bits are a byte. Normally, memory is **byte addressable**, which means that each individual byte has a unique address. Some machines may have a word size that is larger than a single byte. For example, a computer might handle 32-bit words (which means it can manipulate 32 bits at a time through various instructions and it uses 32-bit registers), but still employ a byte-addressable architecture. In this situation, when a word uses multiple bytes, the byte with the lowest address determines the address of the entire word. It is also possible that a computer might be **word addressable**, which means each word (not necessarily each byte) has its own address, but most current machines are byte addressable (even though they have 32-bit or larger words). A memory address is typically stored in a single machine word.

If all this talk about machines using byte addressing with words of different sizes has you somewhat confused, the following analogy may help. Memory is similar to a street full of apartment buildings. Each building (word) has multiple apartments (bytes), and each apartment has its own address. All of the apartments are numbered sequentially (addressed), from 0 to the total number of apartments in the complex minus one. The buildings themselves serve to group the apartments. In computers, words do the same thing. Words are the basic unit of size

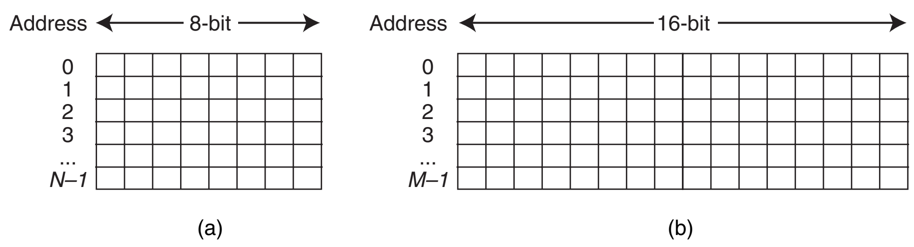


FIGURE 4.4 a) *N* 8-Bit Memory Locations
b) *M* 16-Bit Memory Locations

used in various instructions. For example, you may read a word from or write a word to memory, even on a byte-addressable machine.

If an architecture is byte addressable, and the instruction set architecture word is larger than 1 byte, the issue of **alignment** must be addressed. For example, if we wish to read a 32-bit word on a byte-addressable machine, we must make sure that (1) the word is stored on a natural alignment boundary, and (2) the access starts on that boundary. This is accomplished, in the case of 32-bit words, by requiring the address to be a multiple of 4. Some architectures allow certain instructions to perform unaligned accesses, where the desired address does not have to start on a natural boundary.

Memory is built from random access memory (RAM) chips. (We cover memory in detail in Chapter 6.) Memory is often referred to using the notation length × width (L × W). For example, 4M × 8 means the memory is 4M long (it has $4M = 2^2 \times 2^{20} = 2^{22}$ items) and each item is 8 bits wide (which means that each item is a byte). To address this memory (assuming byte addressing), we need to be able to uniquely identify 2^{22} different items, which means we need 2^{22} different addresses. Because addresses are unsigned binary numbers, we need to count from 0 to $(2^{22} - 1)$ in binary. How many bits does this require? Well, to count from 0 to 3 in binary (for a total of four items), we need 2 bits. To count from 0 to 7 in binary (for a total of eight items), we need 3 bits. To count from 0 to 15 in binary (for a total of 16 items), we need 4 bits. Do you see a pattern emerging here? Can you fill in the missing value for Table 4.1?

The correct answer to the missing table entry is 5 bits. The number of bits required for our 4M memory is 22. Since most memories are *byte* addressable, we say we need N bits to uniquely address each *byte*. In general, if a computer has 2^N addressable units of memory, it requires N bits to uniquely address each unit.

To better illustrate the difference between words and bytes, suppose the 4M × 8 memory referred to in the previous example were word addressable instead of byte addressable and each word were 16 bits long. There are 2^{22} unique bytes, which implies there are $2^{22} \div 2 = 2^{21}$ total words, which would require 21, not 22, bits per address. Each word would require two bytes, but we express the address of the entire word by using the lower byte address.

While most memory is byte addressable and 8 bits wide, memory can vary in width. For example, a 2K × 16 memory holds $2^{11} = 2048$ 16-bit items. This type of memory is typically used on a word-addressable architecture with 16-bit words.

Main memory is usually larger than one RAM chip. Consequently, these chips are combined into a single memory of the desired size. For example, suppose you need to build a 32K × 8 byte-addressable memory and all you have are 2K × 8 RAM chips. You could connect 16 rows of chips together as shown in Figure 4.5.

Total Items	2	4	8	16	32
Total as a Power of 2	2^1	2^2	2^3	2^4	2^5
Number of Bits	1	2	3	4	??

TABLE 4.1 Calculating the Address Bits Required

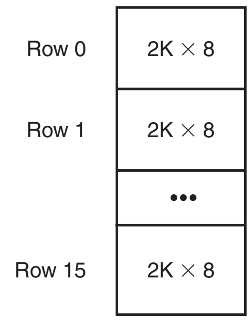


FIGURE 4.5 Memory as a Collection of RAM Chips

Each chip addresses 2K bytes. Addresses for this memory must have 15 bits (there are $32K = 2^5 \times 2^{10}$ bytes to access). But each chip requires only 11 address lines (each chip holds only 2^{11} bytes). In this situation, a decoder is needed to decode either the leftmost or rightmost 4 bits of the address to determine which chip holds the desired data. Once the proper chip has been located, the remaining 11 bits are used to determine the offset on that chip. Whether we use the 4 leftmost or 4 rightmost bits depends on how the memory is interleaved.

A single memory module causes sequentialization of access (only one memory access can be performed at a time). **Memory interleaving**, which splits memory across multiple memory modules (or banks), can be used to help relieve this. With **low-order interleaving**, the low-order bits of the address are used to select the bank; in **high-order interleaving**, the high-order bits of the address are used.

Suppose we have a byte-addressable memory consisting of 8 modules of 4 bytes each, for a total of 32 bytes of memory. We need 5 bits to uniquely identify each byte. Three of these bits are used to determine the module (we have $2^3 = 8$ modules), and the remaining two are used to determine the offset within that module. High-order interleaving, the most intuitive organization, distributes the addresses so that each module contains consecutive addresses, as we see with the 32 addresses in Figure 4.6. Module 0 contains the data stored at addresses 0, 1, 2, and 3; module 1 contains the data stored at addresses 4, 5, 6, and 7; and so on.

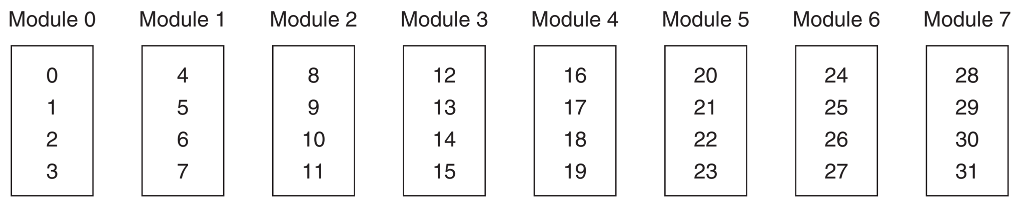


FIGURE 4.6 High-Order Memory Interleaving

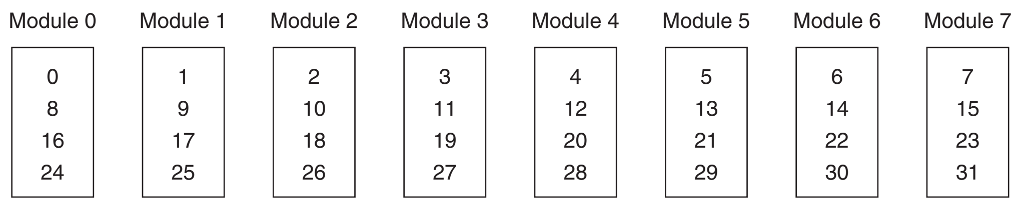


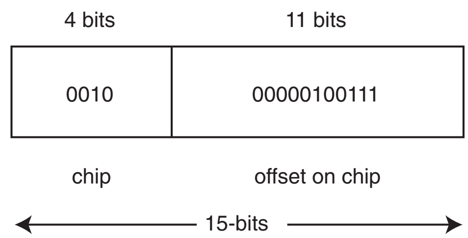
FIGURE 4.7 Low-Order Memory Interleaving

Consider address 3, which in binary (using our required 5 bits), is 00011. High-order interleaving uses the leftmost three bits (000) to determine the module (so the data at address 3 is in module 0). The remaining two bits (11) tell us that the desired data is at offset 3 (11_2 is decimal value 3), the last address in module 0.

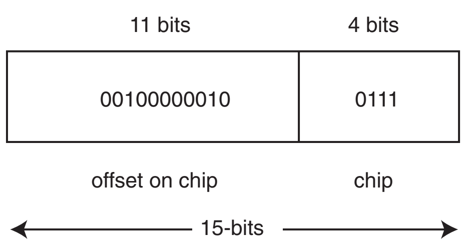
Low-order interleaved memory places consecutive addresses of memory in different memory modules. Figure 4.7 shows low-order interleaving on 32 addresses. In this figure, we see that module 0 now contains the data stored at addresses 0, 8, 16, and 24. To locate address 3 (00011), low-order interleaving uses the rightmost 3 bits to determine the module (which points us to module 3), and the remaining two bits, 00, tell us to look at offset zero within that module. If you check module 3 in Figure 4.7, this is precisely where we find address 3.

With the appropriate buses using low-order interleaving, a read or write using one module can be started before a read or write using another module actually completes (reads and writes can be overlapped). For example, if an array of length 4 is stored in the above example of memory using high-order interleaving (stored at addresses 0, 1, 2, and 3), we are forced to access each array element sequentially, as the entire array is stored in one module. If, however, low-order interleaving is used (and the array is stored in modules 0, 1, 2, and 3 at offset 0 in each), we can access the array elements in parallel because each array element is in a different module.

Let's return to the memory shown in Figure 4.5, a 32K × 8 memory consisting of 16 chips (modules) of size 2K × 8 each. Memory is $32K = 2^5 \times 2^{10} = 2^{15}$ addressable units (in this case, bytes), which means we need 15 bits for each address. Each chip holds $2K = 2^{11}$ bytes, so 11 bits are used to determine the offset on the chip. There are $16 = 2^4$ chips, so we need 4 bits to determine the chip. Consider the address 001000000100111. Using high-order interleaving, we use the 4 leftmost bits to determine the chip, and the remaining 11 as the offset:



The data at address 001000000100111 is stored on chip 2 (0010_2) at offset 39 (00000100111_2). If we use low-order interleaving, the rightmost 4 bits are used to determine the chip:



So the data, using low-order interleaving, is stored on chip 7 (0111_2) at offset 258 (00100000010_2).

Although low-order interleaving allows for concurrent access of data stored sequentially in memory (such as an array or the instructions in a program), high-order interleaving is more intuitive. Therefore, for the remainder of the book, we assume high-order interleaving is being used.

The memory concepts we have covered are very important and appear in various places in the remaining chapters, in particular in Chapter 6, which discusses memory in detail. The key concepts to focus on are: (1) Memory addresses are unsigned binary values (although we often view them as hex values because it is easier), and (2) The number of items to be addressed determines the numbers of bits that occur in the address. Although we could always use more bits for the address than required, that is seldom done because minimization is an important concept in computer design.

4.7 INTERRUPTS

We have introduced the basic hardware information required for a solid understanding of computer architecture: the CPU, buses, control unit, registers, clocks, I/O, and memory. However, there is one more concept we need to cover that deals with how these components interact with the processor: **Interrupts** are events that alter (or interrupt) the normal flow of execution in the system. An interrupt can be triggered for a variety of reasons, including:

- I/O requests
- Arithmetic errors (e.g., division by 0)
- Arithmetic underflow or overflow
- Hardware malfunction (e.g., memory parity error)
- User-defined break points (such as when debugging a program)

- Page faults (this is covered in detail in Chapter 6)
- Invalid instructions (usually resulting from pointer issues)
- Miscellaneous

The actions performed for each of these types of interrupts (called **interrupt handling**) are very different. Telling the CPU that an I/O request has finished is much different from terminating a program because of division by 0. But these actions are both handled by interrupts because they require a change in the normal flow of the program's execution.

An interrupt can be initiated by the user or the system, can be **maskable** (disabled or ignored) or **nonmaskable** (a high priority interrupt that cannot be disabled and must be acknowledged), can occur within or between instructions, may be synchronous (occurs at the same place every time a program is executed) or asynchronous (occurs unexpectedly), and can result in the program terminating or continuing execution once the interrupt is handled. Interrupts are covered in more detail in Section 4.9.2 and in Chapter 7.

Now that we have given a general overview of the components necessary for a computer system to function, we proceed by introducing a simple, yet functional, architecture to illustrate these concepts.

4.8 MARIE

MARIE, a **M**achine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy, is a simple architecture consisting of memory (to store programs and data) and a CPU (consisting of an ALU and several registers). It has all the functional components necessary to be a real working computer. MARIE will help to illustrate the concepts in this and the preceding three chapters. We describe MARIE's architecture in the following sections.

4.8.1 The Architecture

MARIE has the following characteristics:

- Binary, two's complement
- Stored program, fixed word length
- Word (but not byte) addressable
- 4K words of main memory (this implies 12 bits per address)
- 16-bit data (words have 16 bits)
- 16-bit instructions, 4 for the opcode and 12 for the address
- A 16-bit accumulator (AC)
- A 16-bit instruction register (IR)

- A 16-bit memory buffer register (MBR)
- A 12-bit program counter (PC)
- A 12-bit memory address register (MAR)
- An 8-bit input register
- An 8-bit output register

Figure 4.8 shows the architecture for MARIE.

Before we continue, we need to stress one important point about memory. In Chapter 3, we presented a simple memory built using D flip-flops. We emphasize again that each location in memory has a unique address (represented in binary) and each location can hold a value. These notions of the address versus what is actually stored at that address tend to be confusing. To help avoid confusion, visualize a post office. There are post office boxes with various “addresses” or numbers. Inside the post office box, there is mail. To get the mail, the number of the post office box must be known. The same is true for data or instructions that need to be fetched from memory. The contents of any memory address are manipulated by specifying the address of that memory location. We shall see that there are many different ways to specify this address.

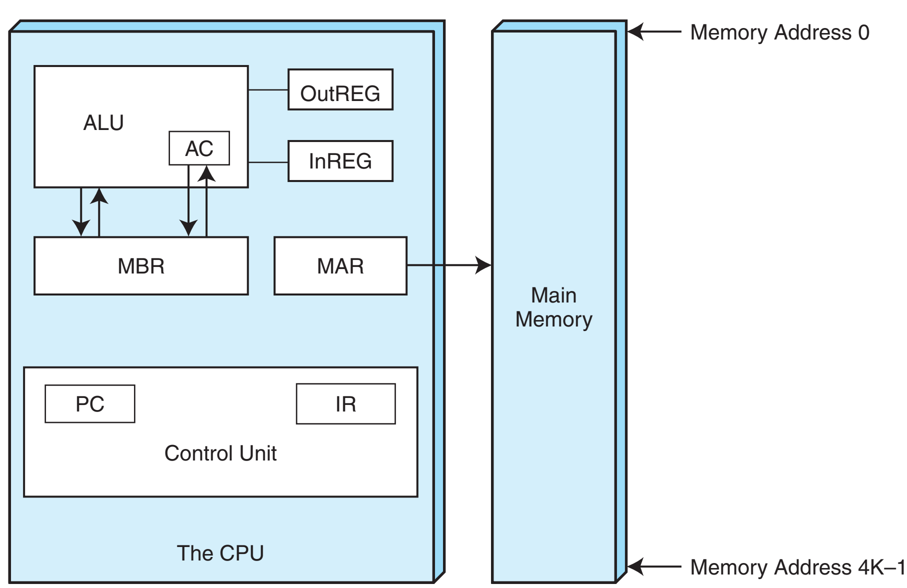


FIGURE 4.8 MARIE’s Architecture

4.8.2 Registers and Buses

Registers are storage locations within the CPU (as illustrated in Figure 4.8). The ALU portion of the CPU performs all of the processing (arithmetic operations, logic decisions, etc.). The registers are used for very specific purposes when programs are executing: They hold values for temporary storage, data that is being manipulated in some way, or results of simple calculations. Many times, registers are referenced implicitly in an instruction, as we see when we describe the instruction set for MARIE in Section 4.8.3.

In MARIE, there are seven registers, as follows:

- **AC:** The **accumulator**, which holds data values. This is a **general-purpose register** and holds data that the CPU needs to process. Most computers today have multiple general-purpose registers.
- **MAR:** The **memory address register**, which holds the memory address of the data being referenced.
- **MBR:** The **memory buffer register**, which holds either the data just read from memory or the data ready to be written to memory.
- **PC:** The **program counter**, which holds the address of the next instruction to be executed in the program.
- **IR:** The **instruction register**, which holds the next instruction to be executed.
- **InREG:** The **input register**, which holds data from the input device.
- **OutREG:** The **output register**, which holds data for the output device.

The MAR, MBR, PC, and IR hold very specific information and cannot be used for anything other than their stated purposes. For example, we could not store an arbitrary data value from memory in the PC. We must use the MBR or the AC to store this arbitrary value. In addition, there is a **status** or **flag register** that holds information indicating various conditions, such as an overflow in the ALU. However, for clarity, we do not include that register explicitly in any figures.

MARIE is a very simple computer with a limited register set. Modern CPUs have multiple general-purpose registers, often called **user-visible registers**, that perform functions similar to those of the AC. Today's computers also have additional registers; for example, some computers have registers that shift data values and other registers that, if taken as a set, can be treated as a list of values.

MARIE cannot transfer data or instructions into or out of registers without a bus. In MARIE, we assume a common bus scheme. Each device connected to the bus has a number, and before the device can use the bus, it must be set to that identifying number. We also have some pathways to speed up execution. We have a communication path between the MAR and memory (the MAR provides the inputs to the address lines for memory so the CPU knows where in

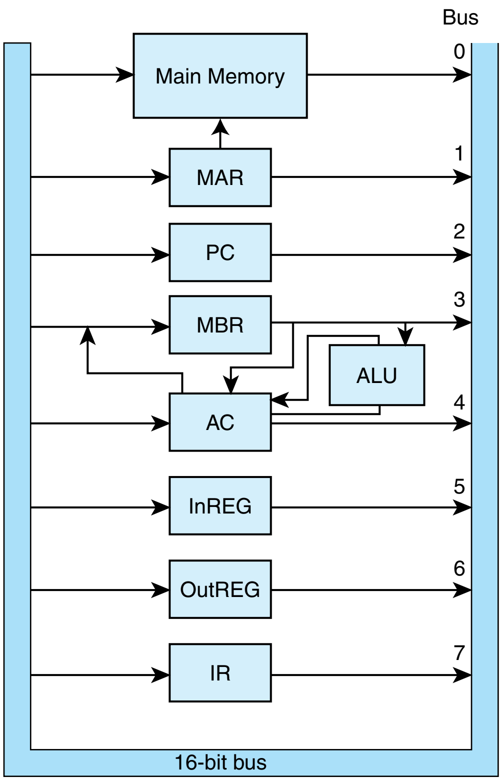


FIGURE 4.9 Datapath in MARIE

memory to read or write), and a separate path from the MBR to the AC. There is also a special path from the MBR to the ALU to allow the data in the MBR to be used in arithmetic operations. Information can also flow from the AC through the ALU and back into the AC without being put on the common bus. The advantage gained using these additional pathways is that information can be put on the common bus in the same clock cycle in which data are put on these other pathways, allowing these events to take place in parallel. Figure 4.9 shows the datapath (the path that information follows) in MARIE.

4.8.3 Instruction Set Architecture

MARIE has a very simple, yet powerful, instruction set. The **instruction set architecture** (**ISA**) of a machine specifies the instructions that the computer can perform and the format for each instruction. The ISA is essentially an interface between the software and the hardware. Some ISAs include hundreds of instructions. We mentioned previously that each instruction for MARIE consists of 16 bits. The most significant 4 bits, bits 12 through 15, make up the **opcode** that specifies the instruction to be executed (which allows for a total of 16 instructions). The least significant 12 bits, bits 0 through 11, form an address, which allows for a maximum memory address of $2^{12}–1$. The instruction format for MARIE is shown in Figure 4.10.

Most ISAs consist of instructions for processing data, moving data, and controlling the execution sequence of the program. MARIE's instruction set consists of the instructions shown in Table 4.2.

The `Load` instruction allows us to move data from memory into the CPU (via the MBR and the AC). All data (which includes anything that is *not* an instruction) from memory must move first into the MBR and then into either the AC or the ALU; there are no other options in this architecture. Notice that the `Load` instruction does not have to name the AC as the final destination; this register is *implicit* in the instruction. Other instructions reference the AC register in a similar fashion. The `Store` instruction allows us to move data from the CPU back to memory. The `Add` and `Subt` instructions add and subtract, respectively, the data value found at address *X* to or from the value in the AC. The data located at address *X* is copied into the MBR where it is held until the arithmetic operation is executed. `Input` and `Output` allow MARIE to communicate with the outside world.

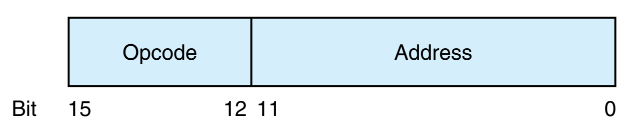


FIGURE 4.10 MARIE's Instruction Format

Instruction Number		Instruction	Meaning
Bin	Hex		
0001	1	Load *X*	Load the contents of address *X* into AC.
0010	2	Store *X*	Store the contents of AC at address *X*.
0011	3	Add *X*	Add the contents of address *X* to AC and store the result in AC.
0100	4	Subt *X*	Subtract the contents of address *X* from AC and store the result in AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate the program.
1000	8	Skipcond	Skip the next instruction on condition.
1001	9	Jump *X*	Load the value of *X* into PC.

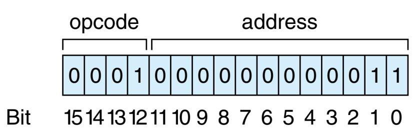
TABLE 4.2 MARIE's Instruction Set

Input and output are complicated operations. In modern computers, input and output are done using ASCII bytes. This means that if you type in the number 32 on the keyboard as input, it is actually read in as the ASCII character "3" followed by "2." These two characters must be converted to the numeric value 32 before they are stored in the AC. Because we are focusing on how a computer works, we are going to assume that a value input from the keyboard is "automatically" converted correctly. We are glossing over a very important concept: How does the computer know whether an I/O value is to be treated as numeric or ASCII, if everything that is input or output is actually ASCII? The answer is that the computer knows through the context of how the value is used. In MARIE, we assume numeric input and output only. We also allow values to be input as decimal and assume there is a "magic conversion" to the actual binary values that are stored. In reality, these are issues that must be addressed if a computer is to work properly.

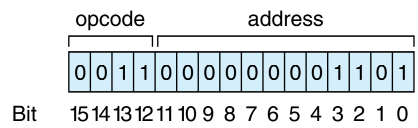
The `Halt` command causes the current program execution to terminate. The `Skipcond` instruction allows us to perform conditional branching (as is done with "while" loops or "if" statements). When the `Skipcond` instruction is executed, the value stored in the AC must be inspected. Two of the address bits (let's assume we always use the two address bits closest to the opcode field, bits 10 and 11) specify the condition to be tested. If the two address bits are 00, this translates to "skip if the AC is negative." If the two address bits are 01 (bit eleven is 0 and bit ten is 1), this translates to "skip if the AC is equal to 0." Finally, if the two address bits are 10 (or 2), this translates to "skip if the AC is greater than 0." By "skip" we simply mean jump over the next instruction. This is accomplished by incrementing the PC by 1, essentially ignoring the following instruction, which is never fetched. The `Jump` instruction, an unconditional branch, also affects the PC. This instruction causes the contents of the PC to be replaced with the value of *X*, which is the address of the next instruction to fetch.

We wish to keep the architecture and the instruction set as simple as possible and yet convey the information necessary to understand how a computer works. Therefore, we have omitted several useful instructions. However, you will see shortly that this instruction set is still quite powerful. Once you gain familiarity with how the machine works, we will extend the instruction set to make programming easier.

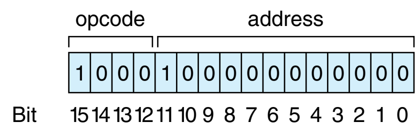
Let's examine the instruction format used in MARIE. Suppose we have the following 16-bit instruction:



The leftmost four bits indicate the opcode, or the instruction to be executed. 0001 is binary for 1, which represents the `Load` instruction. The remaining 12 bits indicate the address of the value we are loading, which is address 3 in main memory. This instruction causes the data value found in main memory, address 3, to be copied into the AC. Consider another instruction:



The leftmost four bits, 0011, are equal to 3, which is the `Add` instruction. The address bits indicate address 00D in hex (or 13 decimal). We go to main memory, get the data value at address 00D, and add this value to the AC. The value in the AC would then change to reflect this sum. One more example follows:



The opcode for this instruction represents the `Skipcond` instruction. Bits ten and eleven (read left to right, or bit eleven followed by bit ten) are 10, indicating a value of 2. This implies a "skip if AC greater than 0." If the value in the AC is less than or equal to zero, this instruction is ignored and we simply go on to the next instruction. If the value in the AC is greater than zero, this instruction causes the PC to be incremented by 1, thus causing the instruction immediately following this instruction in the program to be ignored (keep this in mind as you read the following section on the instruction cycle).

These examples bring up an interesting point. We will be writing programs using this limited instruction set. Would you rather write a program using the commands `Load`, `Add`, and `Halt`, or their binary equivalents 0001, 0011, and 0111? Most people would rather use the instruction name, or **mnemonic**, for the instruction, instead of the binary value for the instruction. Our binary instructions are called **machine instructions**. The corresponding mnemonic instructions are what we refer to as **assembly language instructions**. There is a one-to-one correspondence between assembly language and machine instructions. When we type in an assembly language program (i.e., using the instructions listed in Table 4.2), we need an assembler to convert it to its binary equivalent. We discuss assemblers in Section 4.11.

4.8.4 Register Transfer Notation

We have seen that digital systems consist of many components, including arithmetic logic units, registers, memory, decoders, and control units. These units are interconnected by buses to allow information to flow through the system. The instruction set presented for MARIE in the preceding sections constitutes a set of machine-level instructions used by these components to execute a program. Each instruction appears to be very simplistic; however, if you examine what actually happens at the component level, each instruction involves multiple operations. For example, the `Load` instruction loads the contents of the given memory location into the AC register. But, if we observe what is happening at the component level, we see that multiple "mini-instructions" are being executed. First, the address from the instruction must be loaded into the MAR. Then the data in memory at this location must be loaded into the MBR. Then the MBR must be loaded into the AC. These mini-instructions are called **microoperations** and specify the elementary operations that can be performed on data stored in registers.

The symbolic notation used to describe the behavior of microoperations is called **register transfer notation** (**RTN**) or **register transfer language** (**RTL**). We use the notation *M*[*X*] to indicate the actual data stored at location *X* in memory, and ← to indicate a transfer of information. In reality, a transfer from one register to another always involves a transfer onto the bus from the source register, and then a transfer off the bus into the destination register. However, for the sake of clarity, we do not include these bus transfers, assuming that you understand that the bus must be used for data transfer.

We now present the register transfer notation for each of the instructions in the ISA for MARIE.

Load *X*

Recall that this instruction loads the contents of memory location *X* into the AC. However, the address *X* must first be placed into the MAR. Then the data at location M[MAR] (or address *X*) is moved into the MBR. Finally, this data is placed in the AC.

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

Because the IR must use the bus to copy the value of *X* into the MAR, before the data at location *X* can be placed into the MBR, this operation requires two bus cycles. Therefore, these two operations are on separate lines to indicate they cannot occur during the same cycle. However, because we have a special connection between the MBR and the AC, the transfer of the data from the MBR to the AC can occur immediately after the data is put into the MBR, without waiting for the bus.

Store *X*

This instruction stores the contents of the AC in memory location *X*:

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

Add *X*

The data value stored at address *X* is added to the AC. This can be accomplished as follows:

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

Subt *X*

Similar to `Add`, this instruction subtracts the value stored at address *X* from the accumulator and places the result back in the AC:

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

Input

Any input from the input device is first routed into the InREG. Then the data is transferred into the AC.

```
AC ← InREG
```

Output

This instruction causes the contents of the AC to be placed into the OutREG, where it is eventually sent to the output device.

```
OutREG ← AC
```

Halt

No operations are performed on registers; the machine simply ceases execution of the program.

Skipcond

Recall that this instruction uses the bits in positions 10 and 11 in the address field to determine what comparison to perform on the AC. Depending on this bit combination, the AC is checked to see whether it is negative, equal to 0, or greater than 0. If the given condition is true, then the next instruction is skipped. This is performed by incrementing the PC register by 1.

```
If IR[11-10] = 00 then         {if bits 10 and 11 in the IR are both 0}
   If AC < 0 then PC ← PC + 1
else If IR[11-10] = 01 then    {if bit 11 = 0 and bit 10 = 1}
   If AC = 0 then PC ← PC + 1
else If IR[11-10] = 10 then    {if bit 11 = 1 and bit 10 = 0}
   If AC > 0 then PC ← PC + 1
```

If the bits in positions ten and eleven are both ones, an error condition results. However, an additional condition could also be defined using these bit values.

Jump *X*

This instruction causes an unconditional branch to the given address, *X*. Therefore, to execute this instruction, *X* must be loaded into the PC.

```
PC ← X
```

In reality, the lower or least significant 12 bits of the instruction register (or IR[11–0]) reflect the value of *X*. So this transfer is more accurately depicted as:

```
PC ← IR[11-0]
```

However, we feel that the notation `PC ← X` is easier to understand and relate to the actual instructions, so we use this instead.

Register transfer notation is a symbolic means of expressing what is happening in the system when a given instruction is executing. RTN is sensitive to the datapath, in that if multiple microoperations must share the bus, they must be executed in a sequential fashion, one following the other.

4.9 INSTRUCTION PROCESSING

Now that we have a basic language with which to communicate ideas to our computer, we need to discuss exactly how a specific program is executed. All computers follow a basic machine cycle: the fetch, decode, and execute cycle.

4.9.1 The Fetch–Decode–Execute Cycle

The **fetch–decode–execute cycle** represents the steps that a computer follows to run a program. The CPU fetches an instruction (transfers it from main memory to the instruction register), decodes it (determines the opcode and fetches any data necessary to carry out the instruction), and executes it (performs the operation[s] indicated by the instruction). Notice that a large part of this cycle is spent copying data from one location to another. When a program is initially loaded, the address of the first instruction must be placed in the PC. The steps in this cycle, which take place in specific clock cycles, are listed below. Note that Steps 1 and

2 make up the fetch phase, Step 3 makes up the decode phase, and Step 4 is the execute phase.

1. Copy the contents of the PC to the MAR: `MAR ← PC`.
2. Go to main memory and fetch the instruction found at the address in the MAR, placing this instruction in the IR; increment PC by 1 (PC now points to the next instruction in the program): `IR ← M[MAR]` and then `PC ← PC+1`. (*Note:* Because MARIE is word addressable, the PC is incremented by 1, which results in the next word's address occupying the PC. If MARIE were byte addressable, the PC would need to be incremented by 2 to point to the address of the next instruction, because each instruction would require 2 bytes. On a byte-addressable machine with 32-bit words, the PC would need to be incremented by 4.)
3. Copy the rightmost 12 bits of the IR into the MAR; decode the leftmost 4 bits to determine the opcode, `MAR ← IR[11-0]`, and decode IR[15–12].
4. If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR (and possibly the AC), and then execute the instruction `MBR ← M[MAR]` and execute the actual instruction.

This cycle is illustrated in the flowchart in Figure 4.11.

Note that computers today, even with large instruction sets, long instructions, and huge memories, can execute millions of these fetch–decode–execute cycles in the blink of an eye.

4.9.2 Interrupts and the Instruction Cycle

All computers provide a means for the normal fetch–decode–execute cycle to be interrupted. These interruptions may be necessary for many reasons, including a program error (such as division by 0, arithmetic overflow, stack overflow, or attempting to access a protected area of memory); a hardware error (such as a memory parity error or power failure); an I/O completion (which happens when a disk read is requested and the data transfer is complete); a user interrupt (such as hitting Ctrl-C or Ctrl-Break to stop a program); or an interrupt from a timer set by the operating system (such as is necessary when allocating virtual memory or performing certain bookkeeping functions). All of these have something in common: they interrupt the normal flow of the fetch–decode–execute cycle and tell the computer to stop what it is currently doing and go do something else. They are, naturally, called **interrupts**.

The speed with which a computer processes interrupts plays a key role in determining the computer's overall performance. **Hardware interrupts** can be generated by any peripheral on the system, including memory, the hard drive, the keyboard, the mouse, or even the modem. Instead of using interrupts, processors could poll hardware devices on a regular basis to see if they need anything done. However, this would waste CPU time as the answer would more

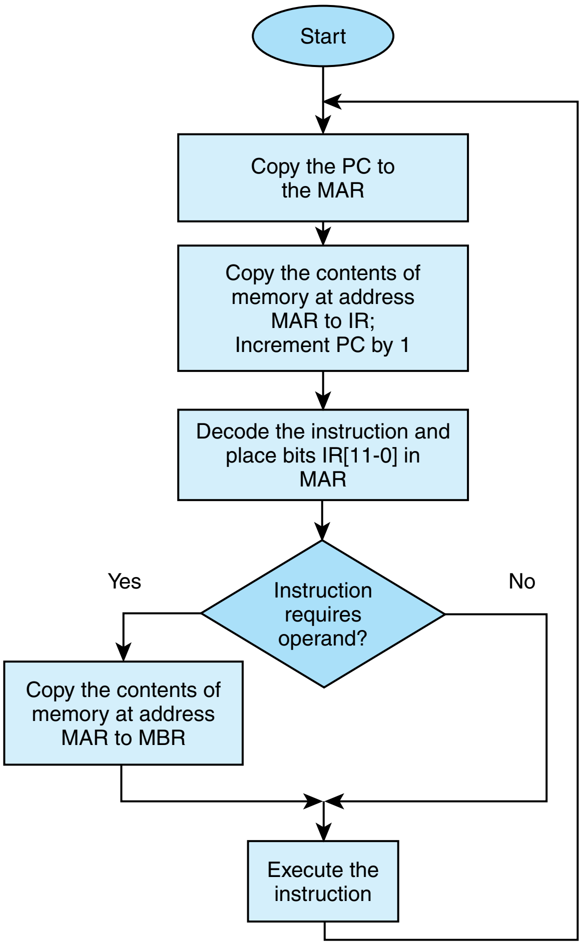


FIGURE 4.11 The Fetch–Decode–Execute Cycle

often than not be "no." Interrupts are nice because they let the CPU know the device needs attention at a particular moment without requiring the CPU to constantly monitor the device. Suppose you need specific information that a friend has promised to acquire for you. You have two choices: call the friend on a regular schedule (polling) and waste his or her time and yours if the information is not ready, or wait for a phone call from your friend once the information has been acquired. You may be in the middle of a conversation with someone else when the phone call "interrupts" you, but the latter approach is by far the more efficient way to handle the exchange of information.

Computers also employ **software interrupts** (also called **traps** or **exceptions**) used by various software applications. Modern computers support both software and hardware interrupts by using **interrupt handlers**. These handlers are simply routines (procedures) that are executed when their respective interrupts are detected. The interrupts, along with their associated **interrupt service routines** (**ISR**s), are stored in an **interrupt vector table**.

How do interrupts fit into the fetch–decode–execute cycle? The CPU finishes execution of the current instruction and checks, at the beginning of every fetch–decode–execute cycle, to see if an interrupt has been generated, as shown in Figure 4.12. Once the CPU acknowledges the interrupt, it must then process the interrupt.

The details of the "Process the Interrupt" block are given in Figure 4.13. This process, which is the same regardless of what type of interrupt has been invoked, begins with the CPU detecting the interrupt signal. Before doing anything else, the system suspends whatever process is executing by saving the program's state and variable information. The device ID or interrupt request number of the device causing the interrupt is then used as an index into the interrupt vector table, which is kept in very low memory. The address of the interrupt service routine (known as its **address vector**) is retrieved and placed into the program counter, and execution resumes (the fetch–decode–execute cycle begins again) within the service routine. After the interrupt service has completed, the system restores the information it saved from the program that was running when the interrupt occurred, and program execution may resume—unless another interrupt is detected, whereupon the interrupt is serviced as described.

It is possible to suspend processing of noncritical interrupts by use of a special interrupt mask bit found in the flag register. This is called **interrupt masking**, and interrupts that can be suspended are called **maskable** interrupts. **Nonmaskable** interrupts cannot be suspended, because to do so, it is possible that the system would enter an unstable or unpredictable state.

Assembly languages provide specific instructions for working with hardware and software interrupts. When writing assembly language programs, one of the

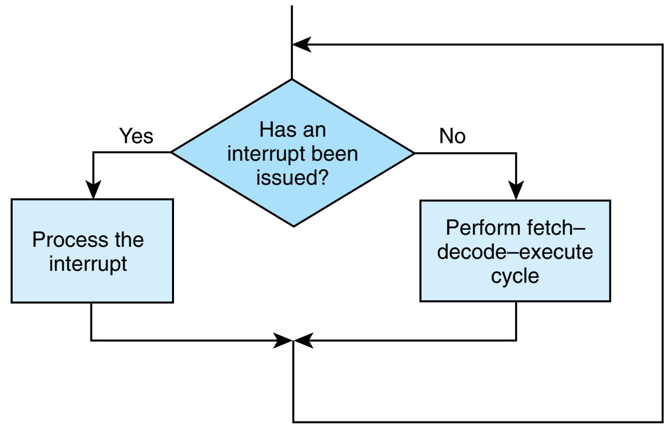


FIGURE 4.12 Fetch–Decode–Execute Cycle with Interrupt Checking

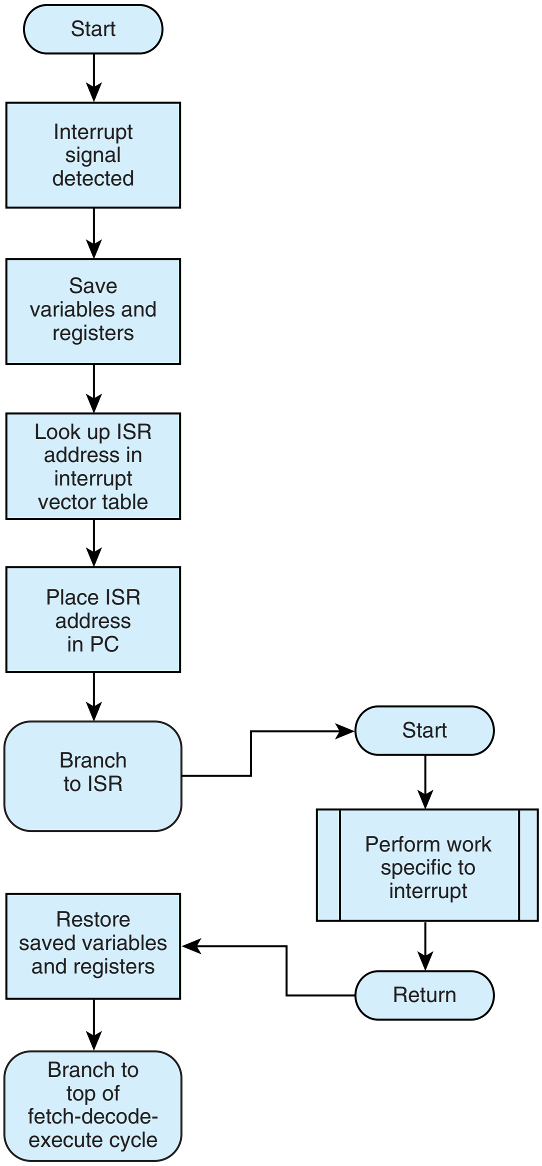


FIGURE 4.13 Processing an Interrupt

most common tasks is dealing with I/O through software interrupts (see Chapter 7 for additional information on interrupt-driven I/O). Indeed, one of the more complicated functions for the novice assembly language programmer is reading input and writing output, specifically because this must be done using interrupts. MARIE simplifies the I/O process for the programmer by avoiding the use of interrupts for I/O.

4.9.3 MARIE's I/O

I/O processing is one of the most challenging aspects of computer system design and programming. Our model is necessarily simplified, and we provide it at this point only to complete MARIE's functionality.

MARIE has two registers to handle input and output. One, the input register, holds data being transferred from an input device into the computer; the other, the output register, holds information ready to be sent to an output device. The timing used by these two registers is very important. For example, if you are entering input from the keyboard and type very fast, the computer must be able to read each character that is put into the input register. If another character is entered into that register before the computer has a chance to process the current character, the current character is lost. It is more likely, because the processor is very fast and keyboard input is very slow, that the processor might read the same character from the input register multiple times. We must avoid both of these situations.

To get around problems like these, MARIE employs a modified type of programmed I/O (discussed in Chapter 7) that places all I/O under the direct control of the programmer. MARIE's output action is simply a matter of placing a value into the OutREG. This register can be read by an output controller that sends it to an appropriate output device, such as a terminal display, printer, or disk. For input, MARIE, being the simplest of simple systems, places the CPU into a wait state until a character is entered into the InREG. The InREG is then copied to the accumulator for subsequent processing as directed by the programmer. We observe that this model provides no concurrency. The machine is essentially idle while waiting for input. Chapter 7 explains other approaches to I/O that make more efficient use of machine resources.

4.10 A SIMPLE PROGRAM

We now present a simple program written for MARIE. In Section 4.12, we present several additional examples to illustrate the power of this minimal architecture. It can even be used to run programs with procedures, various looping constructs, and different selection options.

Our first program adds two numbers together (both of which are found in main memory), storing the sum in memory. (We forgo I/O for now.)

Hex Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0010000100000110	2106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

TABLE 4.3 A Program to Add Two Numbers

Table 4.3 lists an assembly language program to do this, along with its corresponding machine-language program. The list of instructions under the Instruction column constitutes the actual assembly language program. We know that the fetch–decode–execute cycle starts by fetching the first instruction of the program, which it finds by loading the PC with the address of the first instruction when the program is loaded for execution. For simplicity, let's assume our programs in MARIE are always loaded starting at address 100 (in hex).

The list of instructions under the Binary Contents of Memory Address column constitutes the actual machine language program. It is often easier for humans to read hexadecimal as opposed to binary, so the actual contents of memory are displayed in hexadecimal.

This program loads 0023_{16} (or decimal value 35) into the AC. It then adds the hex value FFE9 (decimal −23) that it finds at address 105. This results in a value of 12 in the AC. The `Store` instruction stores this value at memory location 106. When the program is done, the binary contents of location 106 change to 0000000000001100, which is hex 000C, or decimal 12. Figure 4.14 indicates the contents of the registers as the program executes.

The last RTN instruction in part c places the sum at the proper memory location. The statement "decode IR[15–12]" simply means the instruction must be decoded to determine what is to be done. This decoding can be done in software (using a microprogram) or in hardware (using hardwired circuits). These two concepts are covered in more detail in Section 4.13.

Note that there is a one-to-one correspondence between the assembly language and the machine-language instructions. This makes it easy to convert assembly language into machine code. Using the instruction tables given in this chapter, you should be able to hand assemble any of our example programs. For this reason, we look at only the assembly language code from this point on. Before we present more programming examples, however, a discussion of the assembly process is in order.

(a) Load 104

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ⟵ PC	100	------	100	------	------
	IR ⟵ M[MAR]	100	1104	100	------	------
	PC ⟵ PC + 1	101	1104	100	------	------
Decode	MAR ⟵ IR[11–0]	101	1104	104	------	------
	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR ⟵ M[MAR]	101	1104	104	0023	------
Execute	AC ⟵ MBR	101	1104	104	0023	0023

(b) Add 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ⟵ PC	101	1104	101	0023	0023
	IR ⟵ M[MAR]	101	3105	101	0023	0023
	PC ⟵ PC + 1	102	3105	101	0023	0023
Decode	MAR ⟵ IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ⟵ M[MAR]	102	3105	105	FFE9	0023
Execute	AC ⟵ AC + MBR	102	3105	105	FFE9	000C

(c) Store 106

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ⟵ PC	102	3105	102	FFE9	000C
	IR ⟵ M[MAR]	102	2106	102	FFE9	000C
	PC ⟵ PC + 1	103	2106	102	FFE9	000C
Decode	MAR ⟵ IR[11–0]	103	2106	106	FFE9	000C
	(Decode IR[15–12])	103	2106	106	FFE9	000C
Get operand	(not necessary)	103	2106	106	FFE9	000C
Execute	MBR ⟵ AC	103	2106	106	000C	000C
	M[MAR] ⟵ MBR	103	2106	106	000C	000C

FIGURE 4.14 A Trace of the Program to Add Two Numbers

4.11 A DISCUSSION ON ASSEMBLERS

In the program shown in Table 4.3, it is a simple matter to convert from the assembly language instruction `Load 104`, for example, to the machine language instruction 1104 (in hex). But why bother with this conversion? Why not just write in machine code? Although it is very efficient for computers to see these instructions as binary numbers, it is difficult for human beings to understand and program in sequences of 0s and 1s. We prefer words and symbols over long numbers, so it seems a natural solution to devise a program that does this simple conversion for us. This program is called an **assembler**.

4.11.1 What Do Assemblers Do?

An assembler's job is to convert assembly language (using mnemonics) into machine language (which consists entirely of binary values, or strings of 0s and 1s). Assemblers take a programmer's assembly language program, which is really a symbolic representation of the binary numbers, and convert it into binary instructions, or the machine code equivalent. The assembler reads a **source file** (assembly program) and produces an **object file** (the machine code).

Substituting simple alphanumeric names for the opcodes makes programming much easier. We can also substitute **labels** (simple names) to identify or name particular memory addresses, making the task of writing assembly programs even simpler. For example, in our program to add two numbers, we can use labels to indicate the memory addresses, thus making it unnecessary to know the exact memory address of the operands for instructions. Table 4.4 illustrates this concept.

When the address field of an instruction is a label instead of an actual physical address, the assembler still must translate it into a real, physical address in main memory. Most assembly languages allow for labels. Assemblers typically specify formatting rules for their instructions, including those with labels. For example, a label might be limited to three characters and may also be required to occur as the first field in the instruction. MARIE requires labels to be followed by a comma.

Labels are nice for programmers. However, they make more work for the assembler. It must make two passes through a program to do the translation. This

Address		Instruction	
100		Load	X
101		Add	Y
102		Store	Z
103		Halt	
104	X,	0023	
105	Y,	FFE9	
106	Z,	0000	

TABLE 4.4 An Example Using Labels

means the assembler reads the program twice, from top to bottom each time. On the first pass, the assembler builds a set of correspondences called a **symbol table**. For the above example, it builds a table with three symbols: *X*, *Y*, and *Z*. Because an assembler goes through the code from top to bottom, it cannot translate the entire assembly language instruction into machine code in one pass; it does not know where the data portion of the instruction is located if it is given only a label. But after it has built the symbol table, it can make a second pass and "fill in the blanks."

In the above program, the first pass of the assembler creates the following symbol table:

X	104
Y	105
Z	106

It also begins to translate the instructions. After the first pass, the translated instructions would be incomplete as follows:

1			X
3			Y
2			Z
7	0	0	0

On the second pass, the assembler uses the symbol table to fill in the addresses and create the corresponding machine language instructions. Thus, on the second pass, it would know that *X* is located at address 104, and would then substitute 104 for the *X*. A similar procedure would replace the *Y* and *Z*, resulting in:

1	1	0	4
3	1	0	5
2	1	0	6
7	0	0	0

Because most people are uncomfortable reading hexadecimal, most assembly languages allow the data values stored in memory to be specified as binary, hexadecimal, or decimal. Typically, some sort of **assembler directive** (an instruction specifically for the assembler that is not supposed to be translated into machine code) is given to the assembler to specify which base is to be used to interpret the value. We use DEC for decimal and HEX for hexadecimal in MARIE's assembly language. For example, we rewrite the program in Table 4.4 as shown in Table 4.5.

Instead of requiring the actual binary data value (written in HEX), we specify a decimal value by using the directive DEC. The assembler recognizes this directive and converts the value accordingly before storing it in memory. Again, directives are not converted to machine language; they simply instruct the assembler in some way.

Another kind of directive common to virtually every programming language is the **comment delimiter**. Comment delimiters are special characters that tell the

Address		Instruction	
100		Load	X
101		Add	Y
102		Store	Z
103		Halt	
104	X,	DEC	35
105	Y,	DEC	-23
106	Z,	HEX	0000

TABLE 4.5 An Example Using Directives for Constants

assembler (or compiler) to ignore all text following the special character. MARIE's comment delimiter is a front slash ("/"), which causes all text between the delimiter and the end of the line to be ignored.

4.11.2 Why Use Assembly Language?

Our main objective in presenting MARIE's assembly language is to give you an idea of how the language relates to the architecture. Understanding how to program in assembly goes a long way toward understanding the architecture (and vice versa). Not only do you learn basic computer architecture, but you also can learn exactly how the processor works and gain significant insight into the particular architecture on which you are programming. There are many other situations where assembly programming is useful.

Most programmers agree that 10% of the code in a program uses approximately 90% of the CPU time. In time-critical applications, we often need to optimize this 10% of the code. Typically, the compiler handles this optimization for us. The compiler takes a high-level language (such as C++) and converts it into assembly language (which is then converted into machine code). Compilers have been around a long time and in most cases they do a great job. Occasionally, however, programmers must bypass some of the restrictions found in high-level languages and manipulate the assembly code themselves. By doing this, programmers can make the program more efficient in terms of time (and space). This hybrid approach (most of the program written in a high-level language, with part rewritten in assembly) allows the programmer to take advantage of the best of both worlds.

Are there situations in which entire programs should be written in assembly language? If the overall size of the program or response time is critical, assembly language often becomes the language of choice. This is because compilers tend to obscure information about the cost (in time) of various operations and programmers often find it difficult to judge exactly how their compiled programs will perform. Assembly language puts the programmer closer to the architecture and, thus, in firmer control. Assembly language might actually be necessary if the programmer wishes to accomplish certain operations not available in a high-level language.

A perfect example, in terms of both response performance and space-critical design, is found in **embedded systems**. These are systems in which the computer is integrated into a device that is typically not a computer. Embedded systems must be reactive and often are found in time-constrained environments. These systems are designed to perform either a single instruction or a very specific set of instructions. Chances are you use some type of embedded system every day. Consumer electronics (such as cameras, camcorders, cellular phones, PDAs, and interactive games), consumer products (such as washers, microwave ovens, and washing machines), automobiles (particularly engine control and antilock brakes), medical instruments (such as CAT scanners and heart monitors), and industry (for process controllers and avionics) are just a few of the examples of where we find embedded systems.

The software for an embedded system is critical. An embedded software program must perform within very specific response parameters and is limited in the amount of space it can consume. These are perfect applications for assembly language programming. We delve deeper into this topic in Chapter 10.

4.12 EXTENDING OUR INSTRUCTION SET

Even though MARIE's instruction set is sufficient to write any program we wish, there are a few instructions we can add to make programming much simpler. We have 4 bits allocated to the opcode, which implies we can have 16 unique instructions, and we are using only 9 of them. Surely, we can make many programming tasks much easier by adding a few well-chosen instructions to our instruction set. Our new instructions are summarized in Table 4.6.

The `JnS` (jump-and-store) instruction allows us to store a pointer to a return instruction and then proceeds to set the PC to a different instruction. This enables us to call procedures and other subroutines, and then return to the calling point in our code once the subroutine has finished. The `Clear` instruction moves all 0s

Instruction Number (hex)	Instruction	Meaning
0	`JnS` *X*	Store the PC at address *X* and jump to *X* + 1.
A	`Clear`	Put all zeros in AC.
B	`AddI` *X*	Add indirect: Go to address *X*. Use the value at *X* as the actual address of the data operand to add to AC.
C	`JumpI` *X*	Jump indirect: Go to address *X*. Use the value at *X* as the actual address of the location to jump to.
D	`LoadI` *X*	Load indirect: Go to address *X*. Use the value at *X* as the actual address of the operand to load into the AC.
E	`StoreI` *X*	Store indirect: Go to address *X*. Use the value at *X* as the destination address for storing the value in the accumulator.

TABLE 4.6 MARIE's Extended Instruction Set

into the accumulator. This saves the machine cycles that would otherwise be expended in loading a 0 operand from memory.

With the `AddI`, `JumpI`, `LoadI`, and `StoreI` instructions we introduce a different addressing mode. All previous instructions assume the value in the data portion of the instruction is the direct address of the operand required for the instruction. These instructions use the indirect addressing mode. Instead of using the value found at location *X* as the actual address, we use the value found in *X* as a pointer to a new memory location that contains the data we wish to use in the instruction. For example, to execute the instruction `AddI` 400, we first go to location 400. If we find the value 240 stored at location 400, we would go to location 240 to get the actual operand for the instruction. We have essentially allowed for pointers in our language, giving us tremendous power to create advanced data structures and manipulate strings. (We delve more deeply into addressing modes in Chapter 5)

Our six new instructions are detailed below using register transfer notation.

JnS

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

Clear

```
AC ← 0
```

AddI *X*

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

JumpI *X*

```
MAR ← X
MBR ← M[MAR]
PC ← MBR
```

LoadI X

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

StoreI X

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

Table 4.7 summarizes MARIE's entire instruction set.

Let's look at some examples using the full instruction set.

EXAMPLE 4.1 Here is an example using a loop to add five numbers:

```
Address        Instruction
100            Load      Addr  /Load address of first number to be added
101            Store     Next  /Store this address as our Next pointer
102            Load      Num   /Load the number of items to be added
103            Subt      One   /Decrement
104            Store     Ctr   /Store this value in Ctr to control looping
105     Loop,  Load      Sum   /Load the Sum into AC
106            AddI      Next  /Add the value pointed to by location Next
107            Store     Sum   /Store this sum
108            Load      Next  /Load Next
109            Add       One   /Increment by one to point to next address
10A            Store     Next  /Store in our pointer Next
10B            Load      Ctr   /Load the loop control variable
10C            Subt      One   /Subtract one from the loop control variable
10D            Store     Ctr   /Store this new value in loop control variable
10E            Skipcond  000   /If control variable < 0, skip next
                               /instruction
10F            Jump      Loop  /Otherwise, go to Loop
110            Halt            /Terminate program
111     Addr,  Hex       117   /Numbers to be summed start at location 117
112     Next,  Hex       0     /A pointer to the next number to add
113     Num,   Dec       5     /The number of values to add
114     Sum,   Dec       0     /The sum
115     Ctr,   Hex       0     /The loop control variable
116     One,   Dec       1     /Used to increment and decrement by 1
117            Dec       10    /The values to be added together
118            Dec       15
119            Dec       20
11A            Dec       25
11B            Dec       30
```

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

Although the comments are reasonably explanatory, let's walk through Example 4.1. Recall that the symbol table stores [label, location] pairs. The `Load Addr` instruction becomes `Load 111`, because `Addr` is located at physical memory address 111. The value of 117 (the value stored at `Addr`) is then stored in `Next`. This is the pointer that allows us to "step through" the five values we are adding (located at addresses 117, 118, 119, 11A, and 11B). The `Ctr` variable keeps track of how many iterations of the loop we have performed. Because we are checking to see if `Ctr` is negative to terminate the loop, we start by subtracting one from `Ctr`. `Sum` (with an initial value of 0) is then loaded in the AC. The loop begins, using `Next` as the address of the data we wish to add to the AC. The `Skipcond` statement terminates the loop when `Ctr` is negative by skipping the unconditional branch to the top of the loop. The program then terminates when the `Halt` statement is executed.

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
1101	LoadI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← MBR
1110	StoreI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← AC M[MAR] ← MBR

TABLE 4.7 MARIE's Full Instruction Set

Example 4.2 shows how you can use the `Skipcond` and `Jump` instructions to perform selection. Although this example illustrates an `if/else` construct, you can easily modify this to perform an `if/then` structure, or even a `case` (or `switch`) structure.

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 This program demonstrates the use of indirect addressing to traverse and output a string. The string is terminated with a null.

```
Address   Instruction
100       Getch,  LoadI     Chptr    / Load the character found at address Chptr.
101               Skipcond  400      / If AC = 0, skip next instruction.
102               Jump      Outp     / Otherwise, proceed with operation.
103               Halt
104       Outp,   Output             / Output the character.
105               Load      Chptr    / Move pointer to next character.
106               Add       One
107               Store     Chptr
108               Jump      Getch    / Process next character.
109       One,    Hex       0001
10A       Chptr,  Hex       10B      / Pointer to "current" character.
```

```
10B      String, Dec      072  / H   / String definition starts here.
10C              Dec      101  / e
10D              Dec      108  / l
10E              Dec      108  / l
10F              Dec      111  / o
110              Dec      032  / [space]
111              Dec      119  / w
112              Dec      111  / o
113              Dec      114  / r
114              Dec      108  / l
115              Dec      100  / d
116              Dec      033  / !
117              Dec      000  / [null]
END
```

Example 4.3 demonstrates the use of the `LoadI` and `StoreI` instructions by printing a string. Readers who understand the C and C++ programming languages will recognize the pattern: We start by declaring the memory location of the first character of the string and read it until we find a null character. Once the `LoadI` instruction places a null in the accumulator, `Skipcond 400` evaluates to true, and the `Halt` instruction is executed. You will notice that to process each character of the string, we increment the "current character" pointer, `Chptr`, so that it points to the next character to print.

Example 4.4 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 4.4 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 used 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_2, P_1, P_0 that can enable reading from memory or a register and P_5, P_4,

P_3 that can enable writing to a register or memory. 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

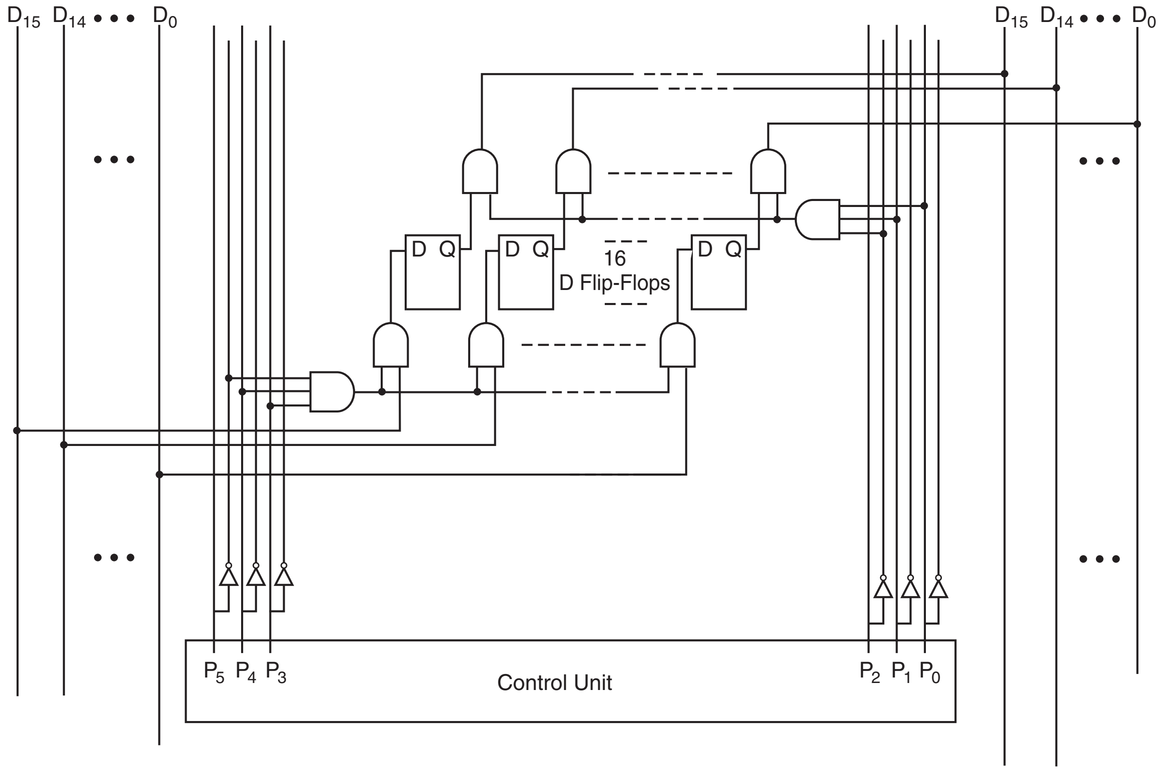


FIGURE 4.15 Connection of MARIE's MBR to the Datapath

ALU Control Signals		
A_0	A_1	ALU Response
0	0	Do Nothing
1	0	AC ← AC + MBR
0	1	AC ← AC – MBR
1	1	AC ← 0 (Clear)

TABLE 4.8 Caption to come

using only two control signals that we'll call A_0 and A_1. These control signals and the ALU response are given in Table 4.8.

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:

$P_2P_1P_0P_3T_0$: `MAR ← X`

$P_4P_3T_1$: `MBR ← M[MAR]`

$A_0P_1P_0P_5T_2$: `AC ← AC + MBR`

C_rT_3: `[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

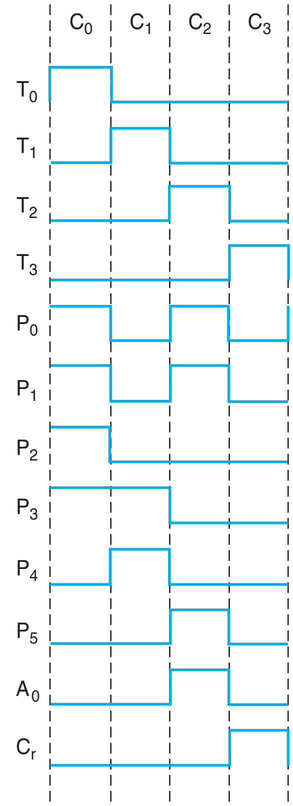


FIGURE 4.16 Timing Diagram for the Microoperations of MARIE's `Add` Instruction

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

There are three essential components common to all hardwired control units: the instruction decoder, the cycle counter, and the control matrix. Depending on the complexity of the system, specialized registers and sets of status flags may be

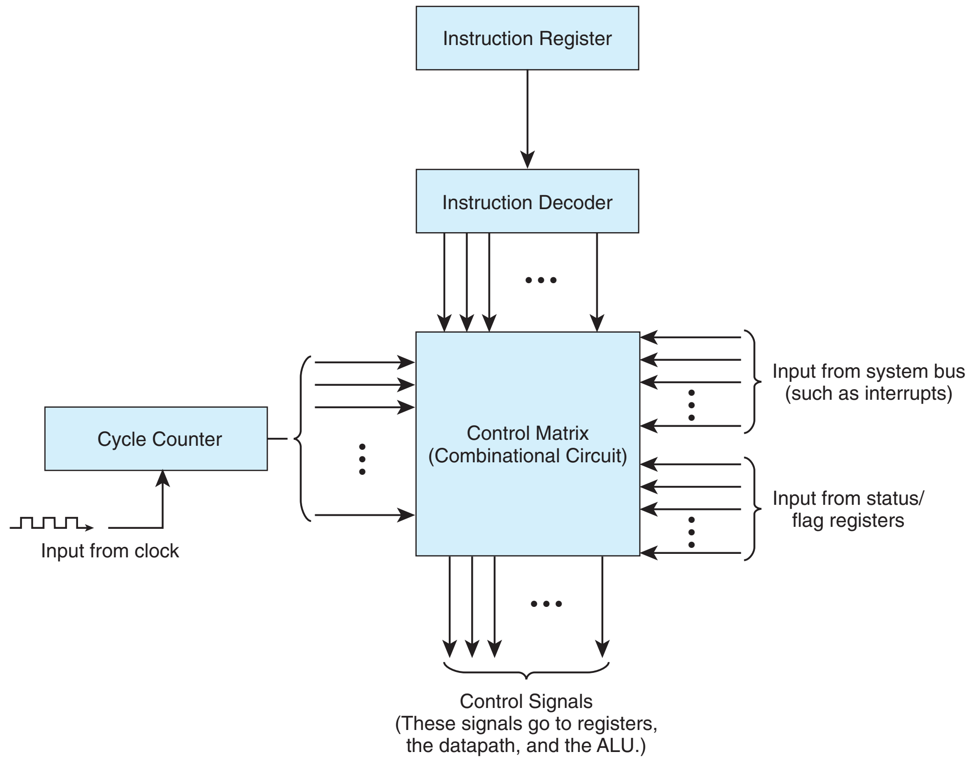


FIGURE 4.17 Hardwired Control Unit

provided as well. Figure 4.17 illustrates a simplified control unit. Let us look at it in detail.

The first essential component is the instruction decoder. Its job is to raise the unique output signal corresponding to the opcode in the instruction register. If we have a four-bit opcode, the instruction decoder could have as many as 16 output signal lines. (Why?) A partial decoder for MARIE's instruction set is shown in Figure 4.18.

The next important component is the control unit's cycle counter. It raises a single, distinct timing signal, T_0, T_1, T_2, …, T_n, for each tick of the system clock. After T_n is reached, the counter cycles back to T_0. The maximum number of microoperations required to carry out any of the instructions in the instruction set determines the number of distinct signals (i.e., the value of n in T_n). MARIE's timer needs to count only up to 7 (T_0 through T_6) to accommodate the `JnS` instruction. (You can verify this statement with a close inspection of Table 4.7.)

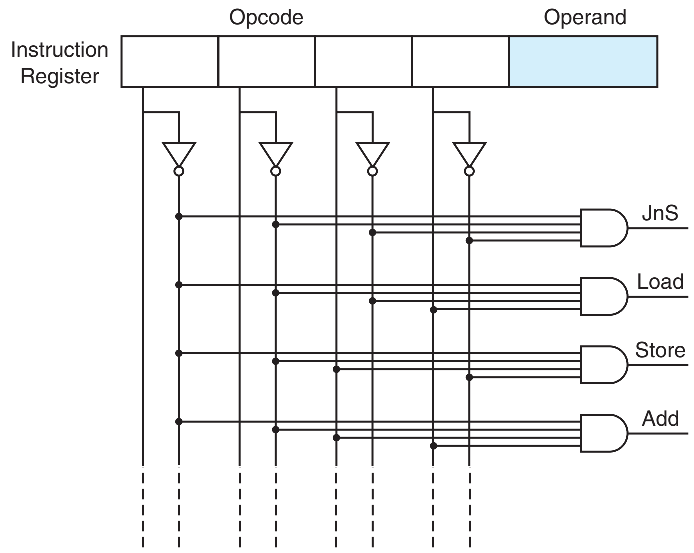


FIGURE 4.18 Partial Instruction Decoder for MARIE's Instruction Set

The sequential logic circuit that provides a repeated series of timing signals is called a ring counter. Figure 4.19 shows one implementation of a ring counter using D flip-flops. Initially, all of the flip-flop inputs are low except for the input to D_0 (because of the inverted OR gate on the other outputs). Thus, in the counter's initial state, output T_0 is energized. At the next tick of the clock, the output of D_0 goes high, causing the input of D_0 to go low (because of the inverted OR gate). T_0 turns off and T_1 turns on. As you can readily see, we have effectively moved a "timing bit" from D_0 to D_1. This bit circulates through the ring of flip-flops until it reaches D_n, unless the ring is first reset by way of the clock reset signal, C_r.

Signals from the counter and instruction decoder are combined within the control matrix to produce the series of signals that result in the execution of microoperations involving the ALU, registers, and datapath.

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. Consider the control unit 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 `Add` instruction is shown in Figure 4.20. You can see how each clock cycle is ANDed

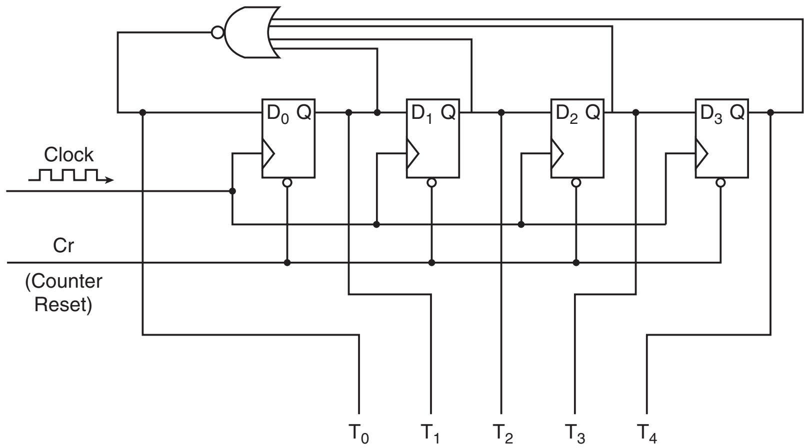


FIGURE 4.19 Ring Counter Using D Flip-Flops

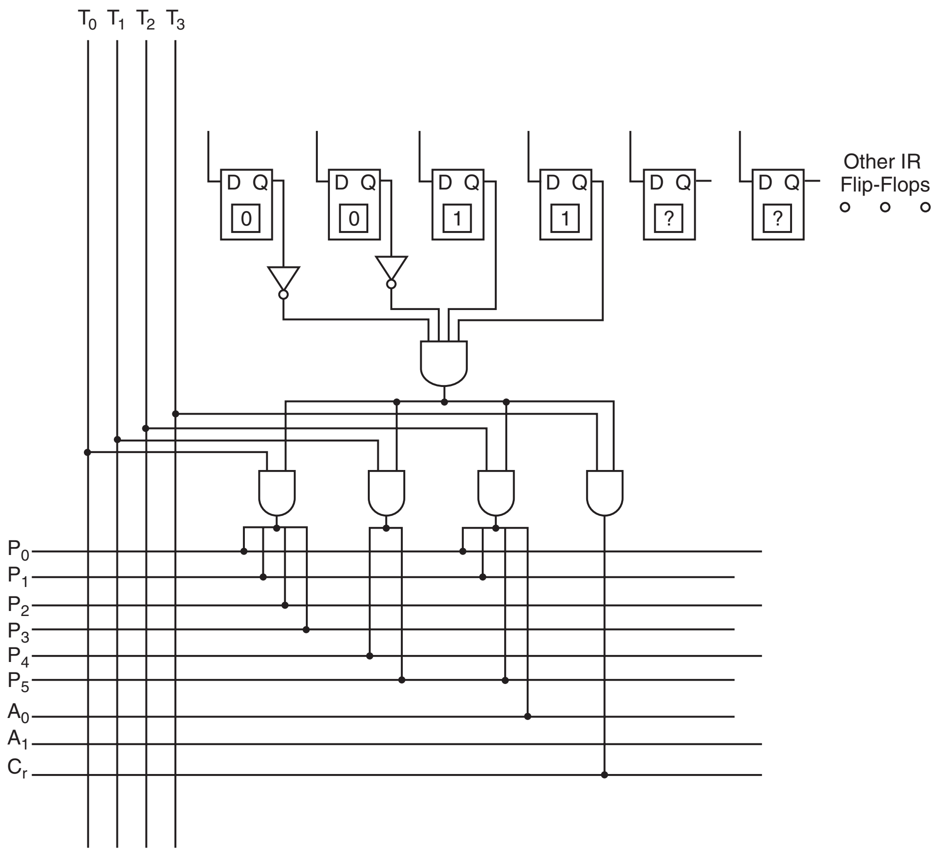


FIGURE 4.20 Combinational Logic for Signal Controls of MARIE's `Add` Instruction

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 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 datapath signals. A generic block diagram of a microprogrammed control unit is shown in Figure 4.21.

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 Figure 4.22.

`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 `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 4.9.

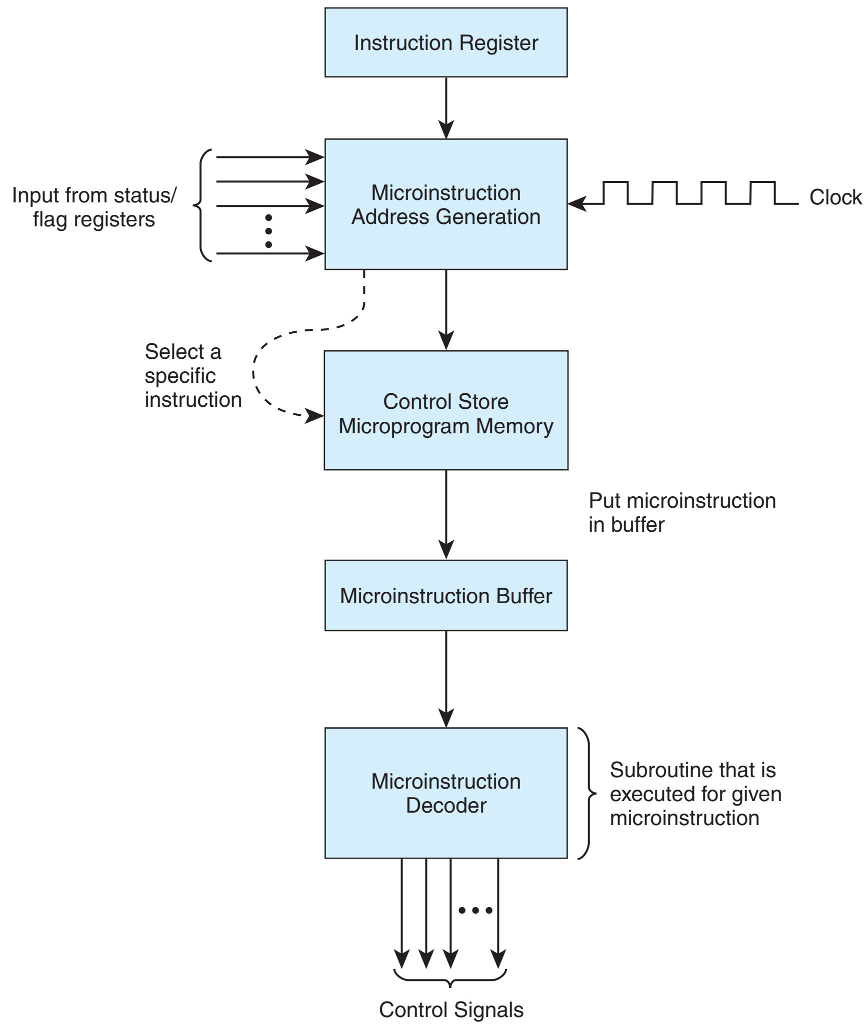


FIGURE 4.21 Microprogrammed Control Unit

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 4.22 MARIE's Microinstruction Format

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01101	MBR ← M[MAR]
00001	AC ← 0	01110	OutREG ← AC
00010	AC ← MBR	01111	PC ← IR[11-0]
00011	AC ← AC - MBR	10000	PC ← MBR
00100	AC ← AC + MBR	10001	PC ← PC + 1
00101	AC ← InREG	10010	If AC = 00
00110	IR ← M[MAR]	10011	If AC > 0
00111	M[MAR] ← MBR	10100	If AC < 0
01000	MAR ← IR[11-0]	10101	If IR[11-10] = 00
01001	MAR ← MBR	10110	If IR[11-10] = 01
01010	MAR ← PC	10111	If IR[11-10] = 10
01011	MAR ← X	11000	If IR[15-12] = MicroOp2[4-1]
01100	MBR ← AC		

TABLE 4.9 Microoperation Codes and Corresponding MARIE RTL

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 4.23 (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 that contain "ifs" 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 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

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 4.23 Selected Statements in MARIE's Microprogram

described in Section 4.13.1. The last instruction for `Add`, at 0101110, 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 4.24 illustrates how this worked, using the AX register as an example.

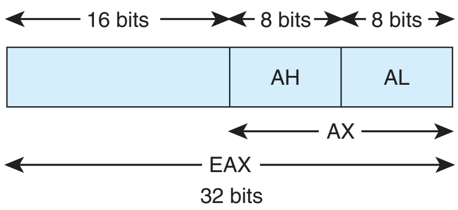


FIGURE 4.24 EAX Register, Broken into Parts

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 a **superscalar** design. This means the CPU had multiple ALUs and could issue more than one instruction per clock cycle (i.e., run instructions in parallel). The Pentium Pro added branch prediction, and the Pentium II added MMX technology (which most will agree was not a huge success) to deal with multimedia. The Pentium III added increased support for 3D graphics (using floating-point instructions). Historically, Intel used a classic CISC approach throughout its processor series. The more recent Pentium II and III used a combined approach, employing CISC architectures with RISC cores that could translate from CISC to RISC instructions. Intel was conforming to the current trend by moving away from CISC and toward RISC.

The seventh-generation family of Intel CPUs introduced the Intel **Pentium IV** (also known as the **Pentium 4**) processor. This processor differs from its predecessors in several different ways, many of which are beyond the scope of this text. Suffice it to say that the Pentium IV processor has clock rates of 1.4 and 1.7 GHz, uses no less than 42 million transistors for the CPU, and implements a **NetBurst microarchitecture**. (The processors in the Pentium family, up to this point, had all been based on the same **microarchitecture**, a term used to describe the architecture below the instruction set.) This type of architecture includes four salient improvements: hyperpipelining, a wider instruction pipeline (pipelining is covered in Chapter 5) to handle more instructions concurrently; a rapid execution engine (the Pentium IV has two arithmetic logic units); an execution trace cache, a cache that holds decoded instructions so if they are used again, they do not have to be decoded again; and a 400 MHz bus. This has made the Pentium IV an extremely useful processor for multimedia applications.

The Pentium IV processor also introduced **hyperthreading (HT)**. **Threads** are tasks that can run independently of one another within the context of the same process. A thread shares code and data with the parent process but has its own resources, including a stack and instruction pointer. Because multiple child threads share with their parent, threads require fewer system resources than if each were a separate process. Systems with more than one processor take advantage of thread processing by splitting instructions so that multiple threads can execute on the processors in parallel. However, Intel's HT enables a single physical processor to simulate two logical (or virtual) processors—the operating system actually sees two processors where only one exists. (To take advantage of HT, the operating system must recognize thread processing.) HT does this through a mix of shared, duplicated, and partitioned chip resources, including registers, math units, counters, and cache memory.

HT duplicates the architectural state of the processor but permits the threads to share main execution resources. This sharing allows the threads to utilize resources that might otherwise be idle (e.g., on a cache miss), resulting in up to a 40% improvement in resource utilization and potential performance gains as high as 25%. Performance gains depend on the application, with computer-intensive applications seeing the most significant gain. Commonplace programs, such as word processors and spreadsheets, are mostly unaffected by HT technology.

What's in a Name?

Intel Corporation makes approximately 80% of the CPUs used in today's microcomputers. It all started with the 4-bit 4004, which in 1971 was the first commercially available microprocessor, or "CPU on a chip." Four years later, Intel's 8-bit 8080 with 6000 transistors was put into the first personal computer, the Altair 8800. As technology allowed more transistors per chip, Intel kept pace by introducing the 16-bit 8086 in 1978 and the 8088 in 1979 (both with approximately 29,000 transistors). These two processors truly started the personal computer revolution, as they were used in the IBM personal computer (later dubbed the XT) and became the industry standard.

The 80186 was introduced in 1980, and although buyers could choose from an 8-bit or a 16-bit version, the 80186 was never used in personal computers. In 1982, Intel introduced the 80286, a 16-bit processor with 134,000 transistors. In fewer than 5 years, over 14 million personal computers were using the 80286 (which most people shortened to simply "286"). In 1985, Intel came out with the first 32-bit microprocessor, the 80386. The 386 multitasking chip was an immediate success, with its 275,000 transistors and 5 million instructions-per-second operating speed. Four years later, Intel introduced the 80486, which had an amazing 1.2 million transistors per chip and operated at 16.9 million instructions per second! The 486, with its built-in math coprocessor, was the first microprocessor to truly rival mainframe computers.

With such huge success and name recognition, why then, did Intel suddenly stop using the 80x86 moniker and switch to *Pentium* in 1993? By this time, many companies were copying Intel's designs and using the same numbering scheme. One of the most successful of these was Advanced Micro Device (AMD). The AMD486 processor had already found its way into many portable and desktop computers. Another was Cyrix with its 486SLC chip. Before introducing its next processor, Intel asked the U.S. Patent and Trademark Office if the company could trademark the name "586." In the United States, numbers cannot be trademarked. (Other countries do allow numbers as trademarks, such as Peugeot's trademark three-digit model numbers with a central zero.) Intel was denied its trademark request and switched the name to *Pentium*. (The astute reader will recognize that *pent* means five, as in *pentagon*.)

It is interesting to note that all of this happened at about the same time as Intel began using its ubiquitous "Intel inside" stickers. It is also interesting that AMD introduced what it called the PR rating system, a method of comparing their x86 processor to Intel's processor. PR stands for "Performance Rating" (not "Pentium Rating" as many people believe) and was intended to guide consumers regarding a particular processor's performance as compared to that of a Pentium.

Intel has continued to manufacture chips using the Pentium naming scheme. The first Pentium chip had 3 million transistors, operated at 25 million instructions per second, and had clock speeds from 60 to 200 MHz. Intel produced many different name variations of the Pentium, including the Pentium MMX in 1997, which improved multimedia performance using the MMX instruction set.

Other manufacturers have also continued to design chips to compete with the Pentium line. AMD introduced the AMD5x86, and later the K5 and K6, to compete with Pentium MMX technology. AMD gave its 5x86 processor a "PR75" rating, meaning this processor was as fast as a Pentium running at 75 MHz. Cyrix introduced the 6x86 chip (or M1) and MediaGX, followed by the Cyrix 6x86MX (M2), to compete with the Pentium MMX.

Intel moved on to the Pentium Pro in 1995. This processor had 5.5 million transistors but had only a slightly larger die than the 4004, which was introduced almost 25 years earlier. The Pentium II (1997) was a cross between the Pentium MMX and the Pentium Pro and contained 7.5 million transistors. AMD continued to keep pace and introduced the K6-2 in 1998, followed by the K6-3. In an attempt to capture more of the low-end market, Intel introduced the Celeron, an entry-level version of the Pentium II with less cache memory.

Intel released the Pentium III in 1999. This chip, housing 9.5 million transistors, used the SSE instruction set (which is an extension to MMX). Intel continued with improvements to this processor by placing cache directly on the core, making caching considerably faster. AMD released the Athlon line of chips in 1999 to compete with the Pentium III. (AMD continues to manufacture the Athlon line to this day.) In 2000, Intel released the Pentium IV, and depending on the particular core, this chip has from 42 to 55 million transistors!

Clearly, changing the name of its processors from the x86 designation to a Pentium-based series has had no negative effects on Intel's success. However, because Pentium is one of the most recognized trademarks in the processor world, industry watchers were surprised when Intel introduced its 64-bit Itanium processor without including *Pentium* as part of the name. Some people believe that this chip name has backfired and their comparison of this chip to a sinking ship has prompted some to call it the *Itanic*.

Intel recently submitted a patent bid to trademark "Intel VIIV." There is considerable speculation as to what VIIV could mean. VI and IV are the Roman numerals for 6 and 4, which could reference 64-bit technology. VIIV might also be representative of Intel's new dual-core chips and could mean 5–2–5, or two Pentium 5 cores.

Although this discussion has given a timeline of Intel's processors, it also shows that, for the past 30 years, Moore's law has held with remarkable accuracy. And we have looked at only Intel and Intel clone processors. There are many other microprocessors we have not mentioned, including those made by Motorola, Zilog, TI, and RCA, to name only a few. With continually increasing power and decreasing costs, there is little wonder that microprocessors have become the most prevalent type of processor in the computer market. Even more amazing is that there is no sign of this trend changing at any time in the near future.

The introduction of the **Itanium** processor in 2001 marked Intel's first 64-bit chip (IA-64). Itanium includes a register-based programming language and a very rich instruction set. It also employs a hardware emulator to maintain backward compatibility with IA-32/x86 instruction sets. This processor has four integer units, two floating-point units, a significant amount of cache memory at four different levels (we study cache levels in Chapter 6), 128 floating-point registers, 128 integer registers, and multiple miscellaneous registers for dealing with efficient loading of instructions in branching situations. Itanium can address up to 16 GB of main memory.

The assembly language of an architecture reveals significant information about that architecture. To compare MARIE's architecture to Intel's architecture, let's return to Example 4.1, the MARIE program that used a loop to add five numbers. Let's rewrite the program in x86 assembly language, as seen in Example 4.5. Note the addition of a `Data` segment directive and a `Code` segment directive.

EXAMPLE 4.5 A program using a loop to add five numbers written to run on a Pentium.

```
       .DATA
Num1   EQU  10               ; Num1 is initialized to 10
       EQU  15               ; Each word following Num1 is initialized
       EQU  20
       EQU  25
       EQU  30
Num    DB   5                ; Initialize the loop counter
Sum    DB   0                ; Initialize the Sum

       .CODE
       LEA  EBX, Num1        ; Load the address of Num1 into EBX
       MOV  ECX, Num         ; Set the loop counter
       MOV  EAX, 0           ; Initialize the sum
       MOV  EDI, 0           ; Initialize the offset (of which number to add)
Start: ADD  EAX, [EBX+EDI*4] ; Add the EBXth number to EAX
       INC  EDI              ; Increment the offset by 1
```

```
        DEC  ECX                 ; Decrement the loop counter by 1
        JG   Start               ; If counter is greater than 0, return to Start
        MOV  Sum, EAX            ; Store the result in Sum
```

We can make this program easier to read (which also makes it look less like MARIE's assembly language) by using the loop statement. Syntactically, the loop instruction resembles a jump instruction, in that it requires a label. This loop can be rewritten as follows:

```
        MOV  ECX, Num                  ; Set the counter
Start:  ADD  EAX, [EBX + EDI * 4]
        INC  EDI
        LOOP Start
        MOV  Sum, EAX
```

The loop statement in x86 assembly is similar to the `do...while` construct in C, C++, or Java. The difference is that there is no explicit loop variable—the ECX register is assumed to hold the loop counter. Upon execution of the loop instruction, the processor decreases ECX by one, and then tests ECX to see if it is equal to 0. If it is not 0, control jumps to `Start`; if it is 0, the loop terminates. The loop statement is an example of the types of instructions that can be added to make the programmer's job easier, but which aren't necessary for getting the job done.

4.14.2 MIPS Architectures

The MIPS family of CPUs has been one of the most successful and flexible designs of its class. The MIPS R3000, R4000, R5000, R8000, and R10000 are some of the many registered trademarks belonging to MIPS Technologies, Inc. MIPS chips are used in embedded systems, in addition to computers (such as Silicon Graphics machines) and various computerized toys (Nintendo and Sony use the MIPS CPU in many of their products). Cisco, a very successful manufacturer of Internet routers, uses MIPS CPUs as well.

The first MIPS ISA was MIPS I, followed by MIPS II through MIPS V. The current ISAs are referred to as MIPS32 (for the 32-bit architecture) and MIPS64 (for the 64-bit architecture). Our discussion in this section focuses on MIPS32. It is important to note that MIPS Technologies made a decision similar to that of Intel—as the ISA evolved, backward compatibility was maintained. And, like Intel, each new version of the ISA included operations and instructions to improve efficiency and handle floating-point values. The new MIPS32 and MIPS64 architectures have significant improvements in VLSI technology and CPU organization. The end result is notable cost and performance benefits over traditional architectures.

Like IA-32 and IA-64, the MIPS ISA embodies a rich set of instructions, including arithmetic, logical, comparison, data transfer, branching, jumping, shifting, and multimedia instructions. MIPS is a **load/store architecture**, which means that all instructions (other than the load and store instructions) must use registers as operands (no memory operands are allowed). MIPS32 has 168 32-bit instructions, but many are similar. For example, there are six different add instructions, all of which add numbers, but they vary in the operands and registers used. This idea of having multiple instructions for the same operation is common in assembly language instruction sets. Another common instruction is the MIPS NOP instruction, which does nothing except eat up time (NOPs are used in pipelining as we see in Chapter 5).

The CPU in a MIPS32 architecture has thirty-two 32-bit general-purpose registers numbered r0 through r31. (Two of these have special functions: r0 is hard-wired to a value of 0 and r31 is the default register for use with certain instructions, which means it does not have to be specified in the instruction itself.) In MIPS assembly, these 32 general-purpose registers are designated \$0, \$1, . . . , \$31. Register 1 is reserved, and registers 26 and 27 are used by the operating system kernel. Registers 28, 29, and 30 are pointer registers. The remaining registers can be referred to by number, using the naming convention shown in Table 4.10. For example, you can refer to register 8 as \$8 or as \$t0.

There are two special purpose registers, HI and LO, which hold the results of certain integer operations. Of course, there is a PC register as well, giving a total of three special-purpose registers.

MIPS32 has thirty-two 32-bit floating-point registers that can be used in single-precision floating-point operations (with double-precision values being stored in even-odd pairs of these registers). There are four special-purpose floating-point control registers for use by the floating-point unit.

Let's continue our comparison by writing the programs from Examples 4.1 and 4.5 in MIPS32 assembly language.

EXAMPLE 4.6

```
        . . .
                .data
        # $t0 = sum
        # $t1 = loop counter Ctr
Value:  .word 10, 15,20,25,30
        Sum = 0
        Ctr = 5
```

Naming Convention	Register Number	Value Put in Register
\$v0-\$v1	2–3	Results, expressions
\$a0-\$a3	4–7	Arguments
\$t0-\$t7	8–15	Temporary values
\$s0-\$s7	16–23	Saved values
\$t8-\$t9	24–25	More temporary values

TABLE 4.10 MIPS32 Register Naming Convention

```
        .text
        .global main          # Declaration of main as a global variable
main:   lw $t0, Sum           # Initialize register containing sum to zero
        lw $t1, Ctr           # Copy Ctr value to register
        la $t2, value         # $t2 is a pointer to current value
while:  blez $t1, end_while  # Done with loop if counter <= 0
        lw $t3, 0($t2)        # Load value offset of 0 from pointer
        add $t0, $t0, $t3     # Add value to sum
        addi $t2, $t2, 4      # Go to next data value
        sub $t1, $t1, 1       # Decrement Ctr
        b while               # Return to top of loop
        la $t4, sum           # Load the address of sum into register
        sw $t0, 0($t4)        # Write the sum into memory location sum
        . . .
```

This is similar to the Intel code in that the loop counter is copied into a register, decremented during each interation of the loop, and then checked to see if it is less than or equal to 0. The register names may look formidable, but they are actually easy to work with once you understand the naming conventions.

If you have enjoyed working with the MARIE simulator and are ready to try your hand at a more complex machine, you will surely find the MIPS Assembler and Runtime Simulator, MARS, to your liking. MARS is a Java-based MIPS R2000 and R3000 simulator designed especially for undergraduate education by Kenneth Vollmar and Pete Sanderson. It provides all the essential MIPS machine functions in a useful and inviting graphical interface. SPIM is another popular MIPS simulator widely used by students and professionals alike. Both of these simulators are freely downloadable and can run on Windows XP and Windows Vista, Mac OS X, Unix, and Linux. For more information see the references at the end of this chapter.

If you examine Examples 4.1, 4.5, and 4.6, you can see that the instructions are quite similar. Registers are referenced in different ways and have different names, but the underlying operations are basically the same. Some assembly languages have larger instruction sets, allowing the programmer more choices for coding various algorithms. But, as we have seen with MARIE, a large instruction set is not absolutely necessary to get the job done.

CHAPTER SUMMARY

This chapter presented a simple architecture, MARIE, as a means to understand the basic fetch–decode–execute cycle and how computers actually operate. This simple architecture was combined with an ISA and an assembly language, with emphasis given to the relationship between these two, allowing us to write programs for MARIE.

The CPU is the principal component in any computer. It consists of a datapath (registers and an ALU connected by a bus) and a control unit responsible for sequencing the operations and data movement and creating the timing signals. All components use these timing signals to work in unison. The I/O subsystem accommodates getting data into the computer and back out to the user.

MARIE is a very simple architecture designed specifically to illustrate the concepts in this chapter without getting bogged down in too many technical details. MARIE has 4K 16-bit words of main memory, uses 16-bit instructions, and has seven registers. There is only one general-purpose register, the AC. Instructions for MARIE use 4 bits for the opcode and 12 bits for an address. Register transfer notation was introduced as a symbolic means for examining what each instruction does at the register level.

The fetch–decode–execute cycle consists of the steps a computer follows to run a program. An instruction is fetched and then decoded, any required operands are then fetched, and finally the instruction is executed. Interrupts are processed at the beginning of this cycle, returning to normal fetch–decode–execute status when the interrupt handler is finished.

A machine language is a list of binary numbers representing executable machine instructions, whereas an assembly language program uses symbolic instructions to represent the numerical data from which the machine language program is derived. Assembly language *is* a programming language, but does not offer a large variety of data types or instructions for the programmer. Assembly language programs represent a lower level method of programming.

You would probably agree that programming in MARIE's assembly language is, at the very least, quite tedious. We saw that most branching must be explicitly performed by the programmer, using jump and branch statements. It is also a large step from this assembly language to a high-level language such as C++ or Ada. However, the assembler is one step in the process of converting source code into something the machine can understand. We have not introduced assembly language with the expectation that you will rush out and become an assembly language programmer. Rather, this introduction should serve to give you a better understanding of machine architecture and how instructions and architectures are related. Assembly language should also give you a basic idea of what is going on behind the scenes in high-level C++, Java, or Ada programs. Although assembly language programs are easier to write for x86 and MIPS than for MARIE, all are more difficult to write and debug than high-level language programs.

Intel and MIPS assembly languages and architectures were introduced (but by no means covered in detail) for two reasons. First, it is interesting to compare the various architectures, starting with a very simple architecture and continuing with much more complex and involved architectures. You should focus on the differences as well as the similarities. Second, although the Intel and MIPS assembly languages looked different from MARIE's assembly language, they are actually quite comparable. Instructions access memory and registers, and there are instructions for moving data, performing arithmetic and logic operations, and branching. MARIE's instruction set is very simple and lacks many of the "programmer friendly" instructions that are present in both Intel and MIPS instruction sets. Intel and MIPS also have more registers than MARIE. Aside from the number of instructions and the number of registers, the languages function almost identically.

FURTHER READING

A MARIE assembly simulator is available on this textbook's home page. This simulator assembles and executes your MARIE programs.

For more detailed information on CPU organization and ISAs, you are referred to the Tanenbaum (2005) and Stallings (2009) books. Mano and Cilett (2006) contains instructional examples of microprogrammed architectures. Wilkes, Renwick, and Wheeler (1958) is an excellent reference on microprogramming.

For more information regarding Intel assembly language programming, check out the Abel (2001), Dandamudi (1998), and Jones (2001) books. The Jones book takes a straightforward and simple approach to assembly language programming, and all three books are quite thorough. If you are interested in other assembly languages, you might refer to Struble (1975) for IBM assembly, Gill, Corwin, and Logar (1987) for Motorola, and SPARC International (1994) for SPARC. For a gentle introduction to embedded systems, try Williams (2000).

If you are interested in MIPS programming, Patterson and Hennessy (2008) give a very good presentation and their book has a separate appendix with useful information. Donovan (1972) and Goodman and Miller (1993) also have good coverage of the MIPS environment. Kane and Heinrich (1992) wrote the definitive text on the MIPS instruction set and assembly language programming on MIPS machines. The MIPS home page also has a wealth of information.

To read more about Intel architectures, please refer to Alpert and Avnon (1993), Brey (2003), Dulon (1998), and Samaras (2001). Perhaps one of the best books on the subject of the Pentium architecture is Shanley (1998). Motorola, UltraSparc, and Alpha architectures are discussed in Circello and co-workers (1995), Horel and Lauterbach (1999), and McLellan (1995), respectively. For a more general introduction to advanced architectures, see Tabak (1994).

If you wish to learn more about the SPIM simulator for MIPS, see Patterson and Hennessy (2008) or the SPIM home page, which has documentation, manuals, and various other downloads. The excellent MARS MIPS Simulator can be downloaded from Vollmar's page at Missouri State University, at http://courses.missouristate.edu/KenVollmar/MARS/. Waldron (1999) is an excellent introduction to RISC assembly language programming and MIPS as well.

REFERENCES

Abel, P. *IBM PC Assembly Language and Programming,* 5th ed. Upper Saddle River, NJ: Prentice Hall, 2001.

Alpert, D., & Avnon, D. "Architecture of the Pentium Microprocessor." *IEEE Micro 13,* April 1993, pp. 11–21.

Brey, B. *Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, and Pentium Pro Processor, Pentium II, Pentium III, and Pentium IV: Architecture, Programming, and Interfacing*, 6th ed. Englewood Cliffs, NJ: Prentice Hall, 2003.

00068_CH04_Null.qxd 10/18/10 12:03 PM Page 258

Circello, J., Edgington, G., McCarthy, D., Gay, J., Schimke, D., Sullivan, S., Duerden, R., Hinds, C., Marquette, D., Sood, L., Crouch, A., & Chow, D. "The Superscalar Architecture of the MC68060." *IEEE Micro 15,* April 1995, pp. 10–21.

Dandamudi, S. P. *Introduction to Assembly Language Programming—From 8086 to Pentium Processors*. New York: Springer Verlag, 1998.

Donovan. J. J. *Systems Programming*. New York: McGraw-Hill, 1972.

Dulon, C. "The IA-64 Architecture at Work." *COMPUTER 31,* July 1998, pp. 24–32.

Gill, A., Corwin, E., & Logar, A. *Assembly Language Programming for the 68000*. Upper Saddle River, NJ: Prentice Hall, 1987.

Goodman, J., & Miller, K. *A Programmer's View of Computer Architecture*. Philadelphia: Saunders College Publishing, 1993.

Horel, T., & Lauterbach, G. "UltraSPARC III: Designing Third Generation 64-Bit Performance." *IEEE Micro 19,* May/June 1999, pp. 73–85.

Jones, W. *Assembly Language for the IBM PC Family*, 3rd ed. El Granada, CA: Scott Jones, Inc., 2001.

Kane, G., & Heinrich, J. *MIPS RISC Architecture,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1992.

Mano, M., & Ciletti, M. *Digital Design*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2006.

McLellan, E. "The Alpha AXP Architecture and 21164 Alpha Microprocessor." *IEEE Micro 15,* April 1995, pp. 33–43.

MIPS home page: *www.mips.com.*

Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. San Mateo, CA: Morgan Kaufmann, 2008.

Samaras, W. A., Cherukuri, N., & Venkataraman, S. "The IA-64 Itanium Processor Cartridge." *IEEE Micro 21,* Jan/Feb 2001, pp. 82–89.

Shanley, T. *Pentium Pro and Pentium II System Architecture.* Reading, MA: Addison-Wesley, 1998.

SPARC International, Inc. *The SPARC Architecture Manual: Version 9*. Upper Saddle River, NJ: Prentice Hall, 1994.

SPIM home page: *www.cs.wisc.edu/~larus/spim.html.*

Stallings, W. *Computer Organization and Architecture*, 8th ed. Upper Saddle River, NJ: Prentice Hall, 2009.

Struble, G. W. *Assembler Language Programming: The IBM System/360 and 370*, 2nd ed. Reading, MA: Addison Wesley, 1975.

Tabak, D. *Advanced Microprocessors*. 2nd ed. New York: McGraw-Hill, 1994.

Tanenbaum, A. *Structured Computer Organization*, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2005.

Vollmar, K., & Sanderson, P. "MARS: An Education-Oriented MIPS Assembly Language Simulator." *ACM SIGCSE Bulletin, 38*:1, March 2006, 239–243.

Waldron, J. *Introduction to RISC Assembly Language*. Reading, MA: Addison Wesley, 1999.

Wilkes, M. V., Renwick, W., & Wheeler, D. J. "The Design of the Control Unit of an Electronic Digital Computer." *Proceedings of IEEE, 105,* 1958, pp. 121–128.

Williams, A. *Microcontroller Projects with Basic Stamps*. Gilroy, CA: R&D Books, 2000.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. What is the function of a CPU?
2. What purpose does a datapath serve?
3. What does the control unit do?
4. Where are registers located and what are the different types?
5. How does the ALU know which function to perform?
6. Why is a bus often a communications bottleneck?
7. What is the difference between a point-to-point bus and a multipoint bus?
8. Why is a bus protocol important?
9. Explain the differences between data buses, address buses, and control buses.
10. What is a bus cycle?
11. Name three different types of buses and where you would find them.
12. What is the difference between synchronous buses and nonsynchronous buses?
13. What are the four types of bus arbitration?
14. Explain the difference between clock cycles and clock frequency.
15. How do system clocks and bus clocks differ?
16. What is the function of an I/O interface?
17. Explain the difference between memory-mapped I/O and instruction-based I/O.
18. What is the difference between a byte and a word? What distinguishes each?
19. Explain the difference between byte addressable and word addressable.
20. Why is address alignment important?
21. List and explain the two types of memory interleaving and the differences between them.
22. Describe how an interrupt works and name four different types.
23. How does a maskable interrupt differ from a nonmaskable interrupt?
24. Why is it that if MARIE has 4K words of main memory, addresses must have 12 bits?
25. Explain the functions of all of MARIE's registers.
26. What is an opcode?
27. Explain how each instruction in MARIE works.
28. How does a machine language differ from an assembly language? Is the conversion one-to-one (one assembly instruction equals one machine instruction)?
29. What is the significance of RTN?
30. Is a microoperation the same thing as a machine instruction?
31. How does a microoperation differ from a regular assembly language instruction?
32. Explain the steps of the fetch–decode–execute cycle.
33. How does interrupt-driven I/O work?
34. Explain how an assembler works, including how it generates the symbol table, what it does with source and object code, and how it handles labels.

35. What is an embedded system? How does it differ from a regular computer?

36. Provide a trace (similar to the one in Figure 4.14) for Example 4.1.

37. Explain the difference between hardwired control and microprogrammed control.

38. What is a stack? Why is it important for programming?

39. Compare CISC machines to RISC machines.

40. How does Intel's architecture differ from MIPS?

41. Name four Intel processors and four MIPS processors.

EXERCISES

1. What are the main functions of the CPU?

2. How is the ALU related to the CPU? What are its main functions?

3. Explain what the CPU should do when an interrupt occurs. Include in your answer the method the CPU uses to detect an interrupt, how it is handled, and what happens when the interrupt has been serviced.

◆ **4.** How many bits would you need to address a 2M × 32 memory if

a) the memory is byte addressable?

b) the memory is word addressable?

5. How many bits are required to address a 4M × 16 main memory if

a) main memory is byte addressable?

b) main memory is word addressable?

6. How many bits are required to address a 1M × 8 main memory if

a) main memory is byte addressable?

b) main memory is word addressable?

7. Suppose we have 4 memory modules instead of 8 in Figures 4.6 and 4.7. Draw the memory modules with the addresses they contain using:

a) High-order interleaving

b) Low-order interleaving.

8. How many 256x8 RAM chips are needed to provide a memory capacity of 4096 bytes?

a) How many bits will each address contain?

b) How many lines must go to each chip?

c) How many lines must be decoded for the chip select inputs? Specify the size of the decoder.

◆ **9.** Suppose that a 2M × 16 main memory is built using 256K × 8 RAM chips and memory is word addressable.

a) How many RAM chips are necessary?

b) How many RAM chips are there per memory word?

c) How many address bits are needed for each RAM chip?

d) How many banks will this memory have?

e) How many address bits are needed for all of memory?

f) If high-order interleaving is used, where would address 14 (which is E in hex) be located?

g) Repeat Exercise 6f for low-order interleaving.

10. Redo Exercise 9 assuming a 16M × 16 memory built using 512K × 8 RAM chips.

11. A digital computer has a memory unit with 24 bits per word. The instruction set consists of 150 different operations. All instructions have an operation code part (opcode) and an address part (allowing for only one address). Each instruction is stored in one word of memory.

a) How many bits are needed for the opcode?

b) How many bits are left for the address part of the instruction?

c) What is the maximum allowable size for memory?

d) What is the largest unsigned binary number that can be accommodated in one word of memory?

12. A digital computer has a memory unit with 32 bits per word. The instruction set consists of 110 different operations. All instructions have an operation code part (opcode) and two address fields: one for a memory address and one for a register address. This particular system includes eight general-purpose, user-addressable registers. Registers may be loaded directly from memory, and memory may be updated directly from the registers. Direct memory-to-memory data movement operations are not supported. Each instruction stored in one word of memory.

a) How many bits are needed for the opcode?

b) How many bits are needed to specify the register?

c) How many bits are left for the memory address part of the instruction?

d) What is the maximum allowable size for memory?

e) What is the largest unsigned binary number that can be accommodated in one word of memory?

13. Assume a 2^{20} byte memory.

◆ **a)** What are the lowest and highest addresses if memory is byte addressable?

◆ **b)** What are the lowest and highest addresses if memory is word addressable, assuming a 16-bit word?

c) What are the lowest and highest addresses if memory is word addressable, assuming a 32-bit word?

14. You and a colleague are designing a brand new microprocessor architecture. Your colleague wants the processor to support 509 different instructions. You do not agree, and would like to have many fewer instructions. Outline the argument for a position paper to present to the management team that will make the final decision. Try to anticipate the argument that could be made to support the opposing viewpoint.

15. Given a memory of 2048 bytes consisting of several 64 × 8 RAM chips, and assuming byte-addressable memory, which of the following seven diagrams indicates the correct way to use the address bits? Explain your answer.

a. ← 10-bit address →

2 bits for chip select	8 bits for address on chip

b. ← 64-bit address →

16 bits for chip select	48 bits for address on chip

c. ← 11-bit address →

6 bits for chip select	5 bits for address on chip

d. ← 6-bit address →

1 bit for chip select	5 bits for address on chip

e. ← 11-bit address →

5 bits for chip select	6 bits for address on chip

f. ← 10-bit address →

4 bits for chip select	6 bits for address on chip

g. ← 64-bit address →

8 bits for chip select	56 bits for address on chip

16. Explain the steps in the fetch–decode–execute cycle. Your explanation should include what is happening in the various registers.

17. Combine the flowcharts that appear in Figures 4.11 and 4.12 so that the interrupt checking appears at a suitable place.

18. Explain why, in MARIE, the MAR is only 12 bits wide and the AC is 16 bits wide. (Hint: Consider the difference between data and addresses.)

19. List the hexadecimal code for the following program (hand assemble it).

```
Hex Address    Label    Instruction
100                     LOAD A
101                     ADD ONE
102                     JUMP S1
103            S2,      ADD ONE
104                     STORE A
105                     HALT
106            S1,      ADD A
107                     JUMP S2
```

```
108            A,       HEX 0023
109            One,     HEX 0001
```

◆ **20.** What are the contents of the symbol table for the preceding program?

21. Consider the MARIE program below.

a) List the hexadecimal code for each instruction.

b) Draw the symbol table.

c) What is the value stored in the AC when the program terminates?

```
Hex Address   Label     Instruction
100           Start,    LOAD A
101                     ADD B
102                     STORE D
103                     CLEAR
104                     OUTPUT
105                     ADDI D
106                     STORE B
107                     HALT
108           A,        HEX 00FC
109           B,        DEC 14
10A           C,        HEX 0108
10B           D,        HEX 0000
```

22. Consider the MARIE program below.

a) List the hexadecimal code for each instruction.

b) Draw the symbol table.

c) What is the value stored in the AC when the program terminates?

```
Hex Address   Label     Instruction
200           Begin,     LOAD Base
201                      ADD Offs
202           Loop,      SUBT One
203                      STORE Addr
204                      SKIPCOND 800
205                      JUMP Done
206                      JUMPI Addr
207                      CLEAR
208           Done,      HALT
209           Base,      HEX 200
20A           Offs,      DEC 9
20B           One,       HEX 0001
20C           Addr,      HEX 0000
```

23. Given the instruction set for MARIE in this chapter, do the following.

Decipher the following MARIE machine language instructions (write the assembly language equivalent):

a) `0010000000000111`

b) `1001000000001011`

c) `0011000000001001`

24. Write the assembly language equivalent of the following MARIE machine language instructions:

a) `0111000000000000`

b) `1011001100110000`

c) `0100111101001111`

25. Write the assembly language equivalent of the following MARIE machine language instructions:

◆ **a)** `0000010111000000`

b) `0001101110010010`

c) `1100100101101011`

26. Write the following code segment in MARIE's assembly language:

```
if X > 1 then
   Y = X + X;
   X = 0;
endif;
Y = Y + 1;
```

27. What are the potential problems (perhaps more than one) with the following assembly language code fragment (implementing a subroutine) written to run on MARIE? The subroutine assumes the parameter to be passed is in the AC and should double this value. The Main part of the program includes a sample call to the subroutine. You can assume this fragment is part of a larger program.

```
Main,  Load   X
       Jump   Sub1
Sret,         Store X
       . . .
Sub1,  Add    X
       Jump   Sret
```

28. Write a MARIE program to evaluate the expression A × B + C × D.

29. Write the following code segment in MARIE assembly language:

```
X = 1;
while X < 10 do
   X = X + 1;
endwhile;
```

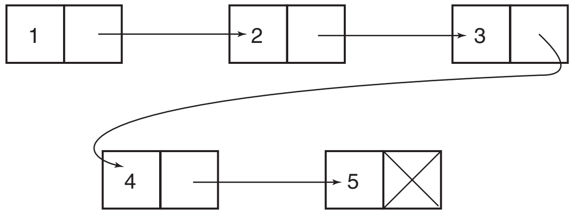
30. Write the following code segment in MARIE assembly language (Hint: Turn the `for` loop into a `while` loop):

```
Sum = 0;
for X = 1 to 10 do
   Sum = Sum + X;
```

31. Write a MARIE program using a loop that multiplies two positive numbers by using repeated addition. For example, to multiply 3 × 6, the program would add 3 six times, or 3 + 3 + 3 + 3 + 3 + 3.

32. Write a MARIE subroutine to subtract two numbers.

33. A linked list is a linear data structure consisting of a set of nodes, where each one except the last one points to the next node in the list. (Appendix A provides more information about linked lists.) Suppose we have the set of 5 nodes shown in the illustration below. These nodes have been scrambled up and placed in a MARIE program as shown below. Write a MARIE program to traverse the list and print the data in order as stored in each node.



MARIE program fragment:

```
Address   Label
(Hex)
00D       Addr,  Hex ????   / Top of list pointer:
                            /  You fill in the address of Node1
00E       Node2, Hex 0032   / Node's data is the character "2"
00F              Hex ????   / Address of Node3
010       Node4, Hex 0034   / Character "4"
011              Hex ????
012       Node1, Hex 0031   / Character "1"
013              Hex ????
014       Node3, Hex 0033   / Character "3"
015              Hex ????
016       Node5, Hex 0035   / Character "5"
017              Hex 0000   / Indicates terminal node
```

34. More registers appear to be a good thing, in terms of reducing the total number of memory accesses a program might require. Give an arithmetic example to support this statement. First, determine the number of memory accesses necessary using MARIE and the two registers for holding memory data values (AC and MBR). Then perform the same arithmetic computation for a processor that has more than three registers to hold memory data values.

35. MARIE saves the return address for a subroutine in memory, at a location designated by the `JnS` instruction. In some architectures, this address is stored in a register, and in many it is stored on a stack. Which of these methods would best handle recursion? Explain your answer. (Hint: Recursion implies many subroutine calls.)

36. Write a MARIE program that performs the three basic stack operations: push, peek, and pop (in that order). In the peek operation, output the value that's on the top of the stack. (If you are unfamiliar with stacks, see Appendix A for more information.)

37. Provide a trace (similar to the one in Figure 4.14) for Example 4.2.

38. Provide a trace (similar to the one in Figure 4.14) for Example 4.3.

39. Suppose we add the following instruction to MARIE's ISA:

```
IncSZ Operand
```

This instruction increments the value with effective address "Operand," and if this newly incremented value is equal to 0, the program counter is incremented by 1. Basically, we are incrementing the operand, and if this new value is equal to 0, we skip the next instruction. Show how this instruction would be written using RTN.

40. Draw the connection of MARIE's PC to the datapath using the format shown in Figure 4.15.

41. The table below provides a summary of MARIE's datapath control signals. Using this information, Table 4.9, and Figure 4.20 as guides draw the control logic for MARIE's `Load` instruction.

Register	Memory	MAR	PC	MBR	AC	IN	OUT	IR
Signals								
$P_2P_1P_0$ (Read) $P_5P_4P_3$ (Write)	000	001	010	011	100	101	110	111

42. The table in Problem 41 provides a summary of MARIE's datapath control signals. Using this information, Table 4.9, and Figure 4.20 as guides draw the control logic for MARIE's `JumpI` instruction.

43. The table in Problem 41 provides a summary of MARIE's datapath control signals. Using this information, Table 4.9, and Figure 4.20 as guides draw the control logic for MARIE's `StoreI` instruction.

44. Suppose some hypothetical system's control unit has a ring (cycle) counter consisting of some number of D flip-flops. This system runs at 1 GHz and has a maximum of 10 microoperations/ instruction.

a) What is the maximum frequency of the output (number of signal pulses) by each flip-flop?

b) How long does it take to execute an instruction that requires only 4 microoperations?

45. Suppose you are designing a hardwired control unit for a very small computerized device. This system is so revolutionary that the system designers have devised an entirely new ISA for it. Because everything is so new, you are contemplating including one or two extra flip-flops and signal outputs in the cycle counter. Why would you want to do this? Why would you not want to do this? Discuss the trade-offs.

46. Building on the idea presented in Problem 45, suppose that MARIE has a hardwired control unit and we decide to add a new instruction that requires 8 clock cycles to execute. (This is one cycle longer than the longest instruction, `JnS`.) Briefly discuss the changes that we would need to make to accommodate this new instruction.

47. Draw the timing diagram for MARIE's `Load` instruction using the format of Figure 4.16.

48. Draw the timing diagram for MARIE's `Subt` instruction using the format of Figure 4.16.

49. Draw the timing diagram for MARIE's `AddI` instruction using the format of Figure 4.16.

50. Using the coding given in Table 4.9, translate into binary the mnemonic microcode instructions given in Figure 4.23 for the fetch-decode cycle (the first nine lines of the table).

51. Continuing from Exercise 50, write the microcode for the jump table for the MARIE instructions for `Jump X`, `Clear`, and `AddI X`. (Use all 1s for the Destination value.)

52. Using Figure 4.23 as a guide, write the binary microcode for MARIE's `Load` instruction. Assume that the microcode begins at instruction line number 0110000_2.

53. Using Figure 4.23 as a guide, write the binary microcode for MARIE's `Add` instruction. Assume that the microcode begins at instruction line number 0110100_2.

54. Would you recommend a synchronous bus or an asynchronous bus for use between the CPU and the memory? Explain your answer.

***55.** Pick an architecture (other than those covered in this chapter). Do research to find out how your architecture deals with the concepts introduced in this chapter, as was done for Intel and MIPS.

True or False

1. If a computer uses hardwired control, the microprogram determines the instruction set for the machine. This instruction set can never be changed unless the architecture is redesigned.

2. A branch instruction changes the flow of information by changing the PC.

3. Registers are storage locations within the CPU itself.

4. A two-pass assembler generally creates a symbol table during the first pass and finishes the complete translation from assembly language to machine instructions on the second.

5. The MAR, MBR, PC, and IR registers in MARIE can be used to hold arbitrary data values.
6. MARIE has a common bus scheme, which means a number of entities share the bus.
7. An assembler is a program that accepts a symbolic language program and produces the binary machine language equivalent, resulting in a one-to-one correspondence between the assembly language source program and the machine language object program.
8. If a computer uses microprogrammed control, the microprogram determines the instruction set for the machine.