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.

4.6 MEMORY ORGANIZATION AND ADDRESSING

We saw an example of a rather small memory in Chapter 3. However, we have no 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), 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. The buildings themselves serve to group the apartments. In computers, words do the same thing. Words are the basic unit of size used in various instructions. For example, you may read a word from or write a word to memory, even on a byte-addressable machine.

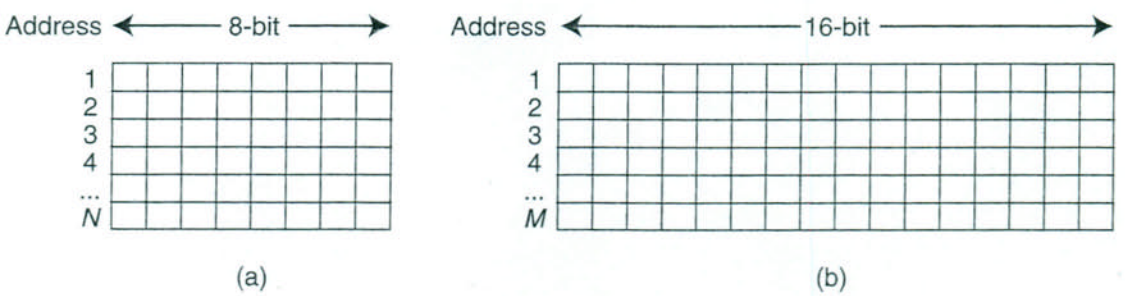

FIGURE 4.4 a) *N* 8-Bit Memory Locations
b) *M* 16-Bit Memory Locations

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 was 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 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 $\times$ width (L $\times$ W). For example, 4M $\times$ 16 means the memory is 4M long (it has $4M = 2^2 \times 2^{20} = 2^{22}$ words) and it is 16 bits wide (each word is 16 bits). The width (second number of the pair) represents the word size. To address this memory (assuming word 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 is 5 bits. In general, if a computer has 2^N addressable units of memory, it requires N bits to uniquely address each byte.

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

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

A single shared memory module causes sequentialization of access. **Memory interleaving**, which splits memory across multiple memory modules (or banks),

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

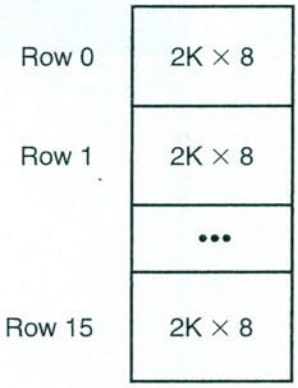

FIGURE 4.5 Memory as a Collection of RAM Chips

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.

High-order interleaving, the more intuitive organization, distributes the addresses so that each module contains consecutive addresses, as we see with the 32 addresses in Figure 4.6.

Low-order interleaved memory places consecutive words of memory in different memory modules. Figure 4.7 shows low-order interleaving on 32 addresses.

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).

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

FIGURE 4.6 High-Order Memory Interleaving

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

FIGURE 4.7 Low-Order Memory Interleaving

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.

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