Chapter

5

Computer Systems Organization

Chapter Introduction

r studying this chapter, you will be able to:

Enumerate the characteristics of the Von Neumann architecture
Describe the components of a random access memory system, including how fetch and store operations work, and the use of cache memory to speed up access time
Diagram the components of a typical arithmetic/logic unit (ALU) and illustrate how the ALU data path operates
Describe the operation of the control unit and explain how it implements the stored program characteristic of the Von Neumann architecture
List and explain the types of instructions in a typical instruction set, and how instructions are commonly encoded
Diagram the organization of a typical Von Neumann machine

Show the sequence of steps, using the book's notation, in the fetch, decode, and execute cycle to perform a typical instruction

5.1 Introduction

Chapter 4 introduced the elementary building blocks of computer systems—transistors, gates, and logic circuits. Although this information is essential to understanding computer hardware—just as knowledge of atoms and molecules is necessary for any serious study of chemistry—it produces a very low-level view of computer systems. Even students who have mastered the material may still ask, "OK, but how do computers *really* work?" Gates and circuits operate on the most elemental of data items, binary 0s and 1s, whereas people reason and work with more complex units of information, such as decimal numbers, character strings, variables, and instructions. To understand how computers process this kind of information, we must look at higher-level components than gates and circuits. We must study computers as collections of **functional units** or subsystems that perform tasks such as instruction processing, information storage, computation, and data transfer. The branch of computer science that studies computers in terms of their major functional units is **computer organization**, and that is the subject of this chapter. This higher-level viewpoint will give us a much better understanding of how a computer really works.

All of the functional units introduced in this chapter are built from the gates and circuits of Chapter 4. However, those elementary components will no longer be visible because we will adopt a different viewpoint, a different perspective, a different **level of abstraction**. This is an extremely important point; as we have said, the concept of abstraction is used throughout computer science. Without it, it would be virtually impossible to study computer design or any other large, complex system.

For example, suppose that system S is composed of a large number of elementary components interconnected in very intricate ways, as shown in Figure 5.1(a). This is equivalent to viewing a computer system as millions or billions of individual gates. For some purposes, it might be necessary to view system S at this level of detail, but for other applications, the amount of complexity could become overwhelming. To deal with this problem, we can redefine the primitives of system S by grouping the elementary components , as shown in Figure 5.1(b), and calling these larger units (A, B, C) the basic building blocks of system S. The three units A, B, and C are then treated as nondecomposable elements whose internal construction is hidden from view. We care only about what functions these components perform and how they interact. This leads to the higher-level system view shown in Figure 5.1(c), which is certainly a great deal simpler than the one shown in Figure 5.1(a), and this is how this chapter approaches the topic of computer hardware. Our primitives are much larger components, similar to A, B, and C, but internally they are still made up of the gates and circuits of Chapter 4.

Figure 5.1 The concept of abstraction

This "abstracting away" of unnecessary detail can be done more than once. For example, at a later point in the study of system S, we might no longer care about the behavior of individual components A, B, and C. Instead, we might want to treat the entire system as a single primitive, nondecomposable entity whose inner workings are no longer important. This leads to the extremely simple system view shown in Figure 5.1(d), a view that we will adopt in later chapters.

Figures 5.1(a), (c), and (d) form what is called a **hierarchy of abstractions**. A hierarchy of abstractions of computer science forms the central theme of this text, and it was initially diagrammed in Figure 1.9. We have already seen this idea in action in Chapter 4, where transistors are grouped into gates and gates into circuits (see Figure 5.2).

Figure 5.2 An example of a hierarchy of abstractions

This process continues into Chapter 5, where we use the addition and comparison circuits of Section 4.4.3 to build an arithmetic unit and use the multiplexer and decoder circuits of Section 4.5 to construct a processor. These higher-level components become our building blocks in all future discussions.

5.2 The Components of a Computer System

There are a huge number of computer systems on the market, manufactured by dozens of different vendors. There are $50 million supercomputers, $1 million mainframes, and $1,000 laptops, as well as tablets and smartphones that may cost less than $100. In addition to size and cost, computers also differ in speed, memory capacity, input/output capabilities, and available software. The hardware marketplace is diverse, multifaceted, and ever changing.

However, in spite of all these differences, virtually every computer in use today is based on a single design. Although a $1 million mainframe, a $1,000 laptop, and a $100 smartphone might not seem to have much in common, they are all based on the same fundamental principles.

The same thing is true of automotive technology. Although a pickup truck, family sedan, and Ferrari racing car do not seem very similar, under the hood they are all constructed from the same basic technology: a gasoline-powered internal combustion engine turning an axle that turns the wheels. (However, electric cars and hybrids represent a radically different approach to automotive engineering. We discuss alternative computer technologies in Section 5.4.) Differences among various models of trucks and cars are not fundamental theoretical differences but simply variations on a theme, such as a bigger engine, a larger carrying capacity, or a more luxurious interior.

The structure and organization of virtually all modern computational devices are based on a single theoretical model called the **Von Neumann architecture**, named after the brilliant mathematician John Von Neumann who proposed it in 1946. (You read about Von Neumann and his enormous contributions to computer science in the historical overview in Section 1.4.)

The Von Neumann architecture is based on the following three characteristics:

- Four major subsystems called *memory, input/output*, the *arithmetic/logic unit (ALU)*, and the *control unit*. These four subsystems are diagrammed in Figure 5.3. The ALU and the control unit are often bundled together in what is called the **central processing unit** or CPU.
- The *stored program* concept, in which the instructions to be executed by the computer are represented as binary values and stored in memory.

- The **sequential execution of instructions**, in which one instruction at a time is fetched from memory and passed to the control unit, where it is decoded and executed.

Figure 5.3 Components of the Von Neumann architecture

This section looks individually at each of the four subsystems that make up the Von Neumann architecture and describes their design and operation. In the following section, we put all these pieces together to show the operation of the overall Von Neumann model.

5.2.1 Memory and Cache

Memory is the functional unit of a computer that stores and retrieves instructions and data. All information stored in memory is represented internally using the binary numbering system described in Section 4.2.

Computer memory uses an access technique called *random access*, and the memory unit is frequently referred to as **random access memory (RAM)**. RAM has the following three characteristics:

- Memory is divided into fixed-size units called cells, and each cell is associated with a unique identifier called an address. These addresses are the unsigned integers 0, 1, 2, ..., MAX.
- All accesses to memory are to a specified address, and we must always fetch or store a complete cell—that is, all the bits in that cell. The cell is the minimum unit of access.
- The time it takes to fetch or store the contents of a single cell is the same for all cells in memory.

A model of a random access memory unit is shown in Figure 5.4. (*Note*: **Read-only memory (ROM)** is a special type of random access memory into which information has been prerecorded during manufacture. This information cannot be modified or removed, only fetched. ROM is used to hold important system instructions and data in a place where a user cannot accidentally or intentionally overwrite them.)

Figure 5.4 Structure of random access memory

As shown in Figure 5.4, the memory unit is made up of cells that contain a fixed number of binary digits. The number of bits per cell is called the cell size or the **memory width**, and it is usually denoted as *W*.

Earlier generations of computers had no standardized value for cell size, and computers were built with values of , 8, 12, 16, 24, 30, 32, 36, 48, and 60 bits. However, computer manufacturers now use a standard cell size of 8 bits, and this 8-bit unit is universally called a *byte*. Thus, the generic term *cell* has become relatively obsolete, and it is more common now to refer to memory bytes as the basic unit. However, keep in mind that this is not a generic term but rather refers to a **memory cell** that contains exactly 8 binary digits.

With a cell size of 8 bits, the largest unsigned integer value that can be stored in a single cell is 11111111, which equals 255—not a very large number. Therefore, computers

with a cell size of often use multiple memory cells to store a single data value. For example, many computers use 2 or 4 bytes (16 or 32 bits) to store one integer, and either 4 or 8 bytes (32 or 64 bits) to store a single real number. This gives the range needed, but at a price. It may take several trips to memory, rather than one, to fetch a single data item. (To solve this problem, all memory units today allow you to fetch and/or store 2, 4, or 8 adjacent memory bytes with a single access. We will have more to say about this feature later in the chapter.)

Each memory cell in RAM is identified by a unique unsigned integer address 0, 1, 2, 3, ... If there are *N* bits available to represent the address of a cell, then the smallest address is 0 and the largest address is a string of *N* 1s:

which is equal to the value . Thus the range of addresses available on a computer is , where *N* is the number of binary digits used to represent an address. This is a total of memory cells. The value is called the **address space** or maximum memory size of the computer. Typical values of *N* in the 1960s and 1970s were 16, 20, 22, and 24. In the 1980s, 1990s, and 2000s, virtually all computers had at least address bits allowing for , or about 4 billion, memory cells. Today, the address space of most computers is at least or about 64 billion bytes, while many processors allow for far more. However, remember that represents the *maximum theoretical* memory size; a computer with *N*address bits does not necessarily come equipped with memory cells. It simply means that its memory can be expanded to . Figure 5.5 gives the value of for several values of *N*.

Figure 5.5

Maximum memory sizes

N	Maximum Memory Size
16	65,536
20	1,048,576
22	4,194,304
24	16,777,216
32	4,294,967,296
40	1,099,511,627,776
50	1,125,899,906,842,624

Because numbers such as 65,536 and 1,048,576 are hard to remember, computer scientists use a convenient shorthand to refer to memory sizes (and other values that are powers of 2). It is based on the fact that the values , , , , and are quite close in magnitude to one thousand, one million, one billion, one trillion, and one quadrillion, respectively. Therefore, the letters K (*kilo*, or thousand), M (*mega*, or million), G (*giga*, or billion), T (*tera*, or trillion), and P (*peta*, or quadrillion) are used to refer to these units.

Thus, a computer with a 16-bit address and bytes of storage would have 64 KB of memory, because . A 64-bit **memory address** would allow, at least theoretically, an address space of , which is approximately 16,000 petabytes!
When dealing with memory, it is important to keep in mind the distinction between an *address* and the *contents* of that address.

The address of this memory cell is 42. The content of cell 42 is the integer value 1. As you will soon see, some instructions operate on addresses, whereas others operate on the contents of an address. A failure to distinguish between these two values can cause confusion about how some instructions behave.

The two basic memory operations are *fetching* and *storing*, and they can be described formally as follows:

- Meaning: Fetch a copy of the contents of the memory cell with the specified *address* and return those contents as the result of the operation. The original contents of the memory cell that was accessed are unchanged. This is termed a **nondestructive fetch**. Given the preceding diagram, the operation Fetch(42) returns the number 1. The value 1 remains in address 42.

- *Store(address, value)*

  Meaning: Store the specified *value* into the memory cell specified by *address*. The previous contents of the cell are lost. This is termed a **destructive store**. The operation Store(42, 2) stores the value 2 into cell 42, overwriting the previous value 1.

Powers of 10

When we talk about volumes of information such as megabytes, gigabytes, and terabytes, it is hard to fathom exactly what those massive numbers mean. Here are some rough approximations (say, to within an order of magnitude) of how much textual information corresponds to each of the storage quantities just introduced, as well as the next few on the scale.

Quantity in Bytes	Base-10 Value	Amount of Textual Information
1 byte		One character
1 kilobyte		One typed page
1 megabyte		Two or three novels
1 gigabyte		A departmental library or a large personal library
1 terabyte		The library of a major academic research university
1 petabyte		All printed material in all libraries in North America

Quantity in Bytes	Base-10 Value	Amount of Textual Information
1 exabyte		All words ever printed throughout human history
1 zettabyte		As of 2014, the total amount of information stored globally on the World Wide Web was about 4–5 zettabytes
1 yottabyte		According to an article in Gizmodo,* a design and technology blog, storing 1 yottabyte of data on hard drives would require 1 million data centers that would fill the states of Rhode Island and Delaware

One of the characteristics of random access memory is that the time to carry out either a fetch or a store operation is the same for all addresses. At current levels of technology, this time, called the **memory access time**, is typically about 5–10 nanoseconds (**nanosecond**). Also note that fetching and storing are allowed only to an entire cell. If we want, for example, to modify a single bit of memory, we first need to fetch the entire cell containing that bit, change the one bit, and then store the entire cell. The cell is the minimum accessible unit of memory.
There is one component of the memory unit shown in Figure 5.4 that we have not yet discussed, the *memory registers*. These two registers are used to implement the fetch and store operations. Both operations require two operands: the address of the cell being accessed and the value, either the value stored by the store operation or the value returned by the fetch operation.

The memory unit contains two special registers whose purpose is to hold these two operands. The **memory address register (MAR)** holds the address of the cell to be accessed. Because the MAR must be capable of holding any address, it must be at least *N* bits wide, where is the address space of the computer.
The **memory data register (MDR)** contains the data value being fetched or stored. We might be tempted to say that the MDR should be *W* bits wide, where W is the cell size. However, as mentioned earlier, on most computers the cell size is only 8 bits, and most data values occupy multiple cells. Thus the size of the MDR is usually a multiple of 8. Typical values of MDR width are 32 and 64 bits, which would allow us to fetch, in a single step, either an integer or a real value, respectively.

Given these two registers, we can describe a little more formally what happens during the fetch and store operations in a random access memory.

- Fetch(address)

1. Load the address into the MAR.
2. Decode the address in the MAR.
3. Copy the contents of that memory location into the MDR.

- Store(address, value)

1. Load the address into the MAR.
2. Load the value into the MDR.
3. Decode the address in the MAR.
4. Store the contents of the MDR into that memory location.

For example, to retrieve the contents of cell 123, we would load the value 123 (in binary, of course) into the MAR and perform a fetch operation. When the operation is done, a copy of the contents of cell 123 would be in the MDR. (We may, for some operations, also automatically fetch the contents of memory cells immediately adjacent to cell 123, e.g., cells 124, 125, ….) To store the value 98 into cell 4, we load a 4 into the MAR and a 98 into the MDR and perform a store. When the operation is completed, the contents of cell 4 will have been set to 98, discarding whatever was there previously.

The operation "Decode the address in the MAR" means that the memory unit must translate the *N*-bit binary address stored in the MAR into the set of signals needed to access that one specific memory cell (or sequence of contiguous memory cells if that feature is allowed). That is, the memory unit must be able to convert the integer value 7, for example, in the MAR into the electronic signals needed to access *only* address 7 from all addresses in the memory unit. This might seem like magic, but it is actually a relatively easy task that applies ideas presented in the previous chapter. We can decode the address in the MAR using a decoder circuit of the type described in Section 4.5 and shown in Figure 4.36. (Remember that a decoder circuit has *N* inputs and outputs numbered . The circuit puts the signal 1 on the output line whose number equals the numeric value on the *N* input lines.) We simply copy the *N* bits in the MAR to the *N* input lines of a decoder circuit. Exactly one of its output lines is ON, and this line's identification number corresponds to the address value in the MAR.

For example, if (the MAR contains 4 bits), then we have 16 addressable cells in our memory, numbered 0000 to 1111 (that is, 0 to 15). We could use a 4-to-16 decoder whose inputs are the 4 bits of the MAR. Each of the 16 output lines is associated with the one memory cell whose address is in the MAR and enables us to fetch or store its contents. This situation is shown in Figure 5.6.

Figure 5.6Organization of memory and the decoding logic

If the MAR contains the 4-bit address 0010 (decimal 2), then only the output line labeled 0010 in Figure 5.6 is ON (that is, carries a value of 1). All others are OFF. The output line 0010 is associated with the unique memory cell that has memory address 2, and the appearance of an ON signal on this line causes the memory hardware to copy the contents of location 2 to the MDR if it is doing a fetch, or to load the contents of the MDR into location 2 if it is doing a store.

The only problem with the memory organization shown in Figure 5.6 is that it does not scale very well. That is, it cannot be used to build a large memory unit. For example, if the number of bits used to represent an address on our computer is 32, a decoder circuit with 32 input lines would have , or more than 4 billion, output lines. That is not a feasible design.

To solve this problem, memories are physically organized into a two-dimensional rather than a one-dimensional organization. In this structure, the 16-byte memory of Figure 5.6would be organized into a two-dimensional 4 × 4 structure, rather than the one-dimensional 16 × 1 organization shown earlier. This two-dimensional layout is shown in Figure 5.7.

Figure 5.7 Two-dimensional memory organization

The memory locations are stored in *row major* order, with bytes 0–3 in row 0, bytes 4–7 in row 1 (01 in binary), bytes 8–11 in row 2 (10 in binary), and bytes 12–15 in row 3 (11 in binary). Each memory cell is connected to two selection lines, one called the *row selection line* and the other called the *column selection line*. When we send a signal down a single row selection line and a single column selection line, only the memory cell located at the *intersection* of these two selection lines carries out a memory fetch or a memory store operation.

How do we choose the correct row and column selection lines to access the proper memory cell? Instead of using one decoder circuit, we use two. The first two binary digits of the addresses in Figure 5.7 are identical to the row number. Similarly, the last two binary digits of the addresses are identical to the column number. Thus, we should no longer view the MAR as being composed of a single 4-bit address, but as a 4-bit address made up of two distinct parts—the leftmost 2 bits, which specify the number of the row containing this cell, and the rightmost 2 bits, which specify the number of the column containing this cell. Each of these 2-bit fields is input to a separate decoder circuit that pulses, respectively, the correct row and column lines to access the desired memory cell.

For example, if the MAR contains the 4-bit value 1101 (a decimal 13), then the two high-order (leftmost) bits 11 are sent to the row decoder, whereas the two low-order (rightmost) bits 01 are sent to the column decoder. The row decoder sends a signal on the line labeled 11 (row 3), and the column decoder sends a signal on the line labeled 01 (column 1). Only the single memory cell in row 3, column 1 becomes active and performs the fetch or store operation. Figure 5.7 shows that the memory cell in row 3, column 1 is the correct one—the cell with memory address 1101.

The two-dimensional organization of Figure 5.7 is far superior to the one-dimensional structure in Figure 5.6 because it can accommodate a much larger number of cells. For example, a memory unit containing 256 MB (bytes) is organized into a 16,384 × 16,384 two-dimensional array. To select any one row or column requires a decoder with 14 input lines and 16,384 output lines. This is a large number of output lines, but it is certainly more feasible to build two 14-to-16,384 decoders than the single 28-to-256 million decoder required for a one-dimensional organization. If necessary, we can continue this process by going to a three-dimensional memory organization, in which the address is broken up into three parts and sent to three separate decoders.
To control whether memory does a fetch or a store operation, our memory unit needs one additional device called a **fetch/store controller**. This unit determines whether we put the contents of a memory cell into the MDR (a fetch operation) or put the contents of the MDR into a memory cell (a store operation). The fetch/store controller is like a traffic officer controlling the direction in which traffic can flow on a two-way street. This memory controller must determine in which direction information flows on the two-way link connecting memory and the MDR. In order to know what to do, this controller receives a signal telling it whether it is to perform a fetch operation (an F signal) or a store operation (an S signal). On the basis of the value of that signal, the controller causes information to flow in the proper direction and the correct memory operation to take place.

A complete model of the organization of a typical random access memory in a Von Neumann architecture is shown in Figure 5.8. Although an actual primary memory unit is far more complex, it follows the basic principles diagrammed in Figure 5.8.

Figure 5.8 Overall RAM organization

Let's complete this discussion by considering how difficult it would be to study the memory unit of Figure 5.8, not at the abstraction level presented in that diagram, but at the gate and circuit level presented in Chapter 4. Let's assume that our memory unit contains cells (4 GB), each byte containing 8 bits. There is a total of about 32 billion bits of storage in this memory unit. A typical memory circuit used to store a single bit generally requires about three gates (1 AND, 1 OR, and 1 NOT) containing seven transistors (3 per AND, 3 per OR, and 1 per NOT). Thus, our 4 GB memory unit (which is actually very modest by today's standards) would contain roughly 96 billion gates and 224 billion transistors, and this does not even include the circuitry required to construct the decoder circuits, the controller, and the MAR and MDR registers! These numbers should help you appreciate the power and advantages of abstraction. Without it, studying a memory unit like the one in Figure 5.8 would be a much more formidable task.

Cache Memory. When Von Neumann created his idealized model of a computer, he described only a single type of memory. Whenever the computer needed an instruction or a piece of data, Von Neumann simply assumed it would get it from RAM using the fetch operation just described. However, as computers became faster, designers noticed that, more and more, the processor was sitting idle waiting for data or instructions to arrive. Processors were executing instructions so quickly that memory access was becoming a bottleneck. (It is hard to believe that a memory unit that can fetch a piece of data in a few billionths of a second can slow anything down, but it does.) As Figure 5.9 shows, during the period from 1980 to 2000, processors increased in performance by a factor of about 3,000, whereas memories became faster by a factor of only about 10. * This led to a huge imbalance between the capabilities of the processor and the capabilities of primary memory, a speed imbalance that has continued to the present time.

Figure 5.9 Changes in memory speed vs. processor speed over time

To solve this problem, designers needed to decrease memory access time to make it comparable with the time needed to carry out an instruction. It is possible to build extremely fast memory units, but they are quite expensive, and providing 32, 64, or 128 GB of this ultra-high-speed memory would make your laptop, tablet, or smartphone prohibitively expensive.

However, computer designers discovered that in order to obtain a significant increase in speed it is not necessary to construct *all* of the memory from expensive, high-speed cells. They observed that when a program fetches a piece of data or an instruction, there is a high likelihood that

1. It will access that same instruction or piece of data in the very near future.
2. It will soon access the instructions or data that are located near that piece of data, where "near" means an address whose numerical value is close to this one.
   Simply stated, this observation, called the **principle of locality**, says that when the computer uses something, it will probably use it again very soon, and it will probably use the "neighbors" of this item very soon. (Think about a loop in an algorithm that keeps repeating the same instruction sequence over and over.) To exploit this observation, the first time that the computer references a piece of data, it should copy that data from regular RAM memory to a special, high-speed memory unit called **cache memory** (pronounced "cash," from the French word *cacher*, meaning "to hide"). It should also copy the contents of memory cells located near this item into the cache. A cache is typically 5–10 times faster than RAM but much smaller—on the order of a few megabytes rather than a few dozen gigabytes. This limited size is not a problem because the computer does not keep all of the data there, just those items that were accessed most recently and that, presumably, will be needed again immediately. The organization of the "two-level memory hierarchy" is shown in Figure 5.10.

Figure 5.10 Two-level memory hierarchy

When the computer needs a piece of information, it does not immediately do the memory fetch operation described earlier. Instead, it carries out the following three steps:

1. Look first in cache memory to see whether the information is there. If it is, then the computer can access it at the higher speed of the cache.
2. If the desired information is not in the cache, then access it from RAM at the slower speed, using the fetch operation described earlier.
3. Copy the data just fetched into the cache along with the k immediately following memory locations. If the cache is full, then copy the new data into the cache so as to overwrite some of the older items that have not recently been accessed. (The assumption is that we will not need them again for a while.)
   This algorithm significantly reduces the average time to access information. For example, assume that the average access time of our RAM is 10 nsec, whereas the average access time of the cache is 2 nsec. Furthermore, assume that the information we need is in the cache 70% of the time, a value called the **cache hit rate**. In this situation, 70% of the time we get what we need in 2 nsec, and 30% of the time we have wasted that 2 nsec because the information is not in the cache and must be obtained from RAM, which will take 10 nsec. Our overall average access time will now be

which is a 50% reduction in access time from the original value of 10 nsec. A higher cache hit rate can lead to even greater savings.

A good analogy to cache memory is a home refrigerator. Without one we would have to go to the grocery store every time we needed an item; this corresponds to slow, regular memory access. Instead, when we go to the store we buy not only what we need now but also what we think we will need in the near future, and we put those items into our refrigerator. Now, when we need something, we first check the refrigerator. If it is there, we can get it at a much higher rate of speed. We only need to go to the store when the food item we want is not there.

Caches are found on every modern computer system, and they are a significant contributor to the higher computational speeds achieved by new machines. (In fact, most modern processors have not just a single cache but two, three, or even four levels of cache memory, each serving a different purpose.) Even though the formal Von Neumann model contained only a single memory unit, most computers built today have a multilevel hierarchy of random access memory.

5.2.2 Input/Output and Mass Storage

The **input/output (I/O)** units are the devices that allow a computer system to communicate and interact with the outside world as well as store information for the long term. The random access memory described in the previous section is **volatile memory**—the information disappears when the power is turned off. Without some type of long-term, **nonvolatile memory**, information could not be saved between shutdowns of the machine. Nonvolatile storage is the role of **mass storage systems** such as disks, flash drives, and tapes. (Today, a good deal of long-term data storage is no longer on local I/O devices but on remote data servers in special locations called **data centers**. This type of *cloud storage* is discussed in Chapter 7.)

Of all the components of a Von Neumann machine, the I/O and mass storage subsystems are the most ad hoc and the most variable. Unlike the memory unit, I/O does not adhere to a single well-defined theoretical model. On the contrary, there are dozens of different I/O and mass storage devices manufactured by dozens of different companies and exhibiting many alternative organizations, making generalizations difficult. However, two important principles transcend the device-specific characteristics of particular vendors—*I/O access methods* and *I/O controllers*.

Input/output devices come in two basic types: those that represent information in *human-readable* form for human consumption and those that store information in *machine-readable* form for access by a computer system. The former includes such well-known I/O devices as keyboards, both physical and virtual, screens, and printers. The latter group of devices includes flash memory, hard drives, DVDs, and streaming tapes. Mass storage devices themselves come in two distinct forms: **direct access storage devices (DASDs)** and **sequential access storage devices (SASDs)**.

Our discussion on random access memory in Section 5.2.1 described the fundamental characteristics of *random* access:

1. Every memory cell has a unique address.
2. It takes the same amount of time to access every cell.

A *direct access storage device (DASD)* is one in which requirement number 2, equal access time, has been eliminated. That is, in a direct access storage device, every unit of information still has a unique address, but the time needed to access that information depends on its physical location and the current state of the device.

The best examples of DASDs are the types of disks listed earlier: hard drives, DVDs, and so on. A magnetic disk stores information in units called **sectors**, each of which contains an address and a data block containing a fixed number of bytes, illustrated in Figure 5.11. (*Note*: Flash memory devices and solid-state drives do not have rotating disks or the moveable read/write arms described in this section. They operate more like

nonvolatile primary storage and should be considered as random access mass storage devices rather than direct access storage.)

Figure 5.11 **Disk sector**

A fixed number of these sectors are placed in a concentric circle on the surface of the disk, called a **track**, shown in Figure 5.12.

Figure 5.12 **A single disk track**

Finally, the surface of the disk contains many tracks, and there is a single *read/write head* that can be moved in or out to be positioned over any track on the disk surface. The entire disk rotates at high speed under the read/write head. The overall organization of a typical rotating disk is shown in Figure 5.13.

Figure 5.13 **Overall organization of a typical rotating disk**

The access time to any individual sector of the disk is made up of three components: seek time, latency, and transfer time. **Seek time** is the time needed to position the read/write head over the correct track; **latency** is the time for the beginning of the desired sector to rotate under the read/write head; and **transfer time** is the time for the entire sector to pass under the read/write head and have its contents read into or written from memory. These values depend on the specific sector being accessed and the current position of the read/write head. Let's assume a disk drive with the following physical characteristics :

to move to an adjacent track (i.e., moving from track i to either track $i + 1$ or $i - 1$)
(numbered 0 to 999)

The access time for this disk can be determined as follows.

1. Seek Time	(no arm movement) (move from track 0 to track 999) (assume that on average, the read/write head must move about 300 tracks)
2. Latency	(sector is just about to come under the read/write head) (we just missed the first bit of the sector and must wait one full revolution) (one-half a revolution)
3. Transfer	(the time for one sector, or 1/64th of a track, to pass under the read/write head; this time will be the same for all sectors)

Figure 5.14 summarizes these access time computations.

Figure 5.14

Example disk access times in milliseconds

	Best	Worst	Average
Seek Time	0	19.98	6
Latency	0	8.33	4.17
Transfer	0.13	0.13	0.13
Total	0.13	28.44	10.3

The best-case time and the worst-case time to fetch or store a sector on the disk differ by a factor of more than 200, that is, 0.13 msec versus 28.44 msec. The average access time is about 10 msec, a typical value for current hard drive technology. However, even as direct access storage devices get faster and faster, this difference in best case versus worst case access times will remain large. This is the fundamental characteristic of all direct access storage devices, not just disks: They enable us to specify the address of the desired unit of data and go directly to that data item, but they cannot provide a uniform access time. Today, there is an enormous range of direct access storage devices in the marketplace, from small hard drives that hold a few gigabytes, to DVDs that can store hundreds of gigabytes, to massive online storage devices capable of recording and accessing terabytes or even petabytes of data.

The second type of mass storage device uses the old access technique called *sequential access*. A sequential access storage device (SASD) does not require that all units of data be identifiable via unique addresses. To find any given data item, we must search all data sequentially, repeatedly asking the question, “Is this what I’m looking for?” If not, we move on to the next unit of data and ask the question again. Eventually we find what we are looking for or come to the end of the data.

A sequential access storage device behaves just like the old audio cassette tapes of the 1980s and 1990s. To locate a specific song, we run the tape for a while and then stop and listen. This process is repeated until we find the desired song or come to the end of the tape. In contrast, a direct access storage device behaves like a music DVD that numbers the songs and allows you to select any one. (The song number is the address.) Direct access storage devices are generally much faster at accessing individual pieces of information, and that is why they are much more widely used for mass storage. However, sequential access storage devices can be useful in specific situations, such as sequentially copying the entire contents of memory or of a disk drive. This backup operation fits the SASD model well, and *streaming tape backup units* are common storage devices on computer systems.

One of the fundamental characteristics of many (although not all) I/O devices is that they are very, very *slow* when compared with other components of a computer. For example, a typical memory access time is about 10 nsec. The time to complete the I/O operation “locate and read one disk sector” was shown in the previous example to be about 10 msec.

Units such as nsec (billionths of a second), μsec (millionths of a second), and msec (thousandths of a second) are so small compared with human time scales that it is

sometimes difficult to appreciate the immense difference between values like 10 nsec and 10 msec. The difference between these two quantities is a factor of 1,000,000, that is, 6 orders of magnitude. Consider that this is the same order of magnitude difference as between 1 mile and 40 complete revolutions of the Earth's equator, or between 1 day and 30 centuries!

It is not uncommon for I/O operations such as displaying an image on a screen or printing a page on a printer to be 3, 4, 5, or even 6 orders of magnitude slower than any other aspect of computer operation. If there isn't something in the design of a computer to account for this difference, components that operate on totally incompatible time scales will be trying to talk to each other, which will produce enormous inefficiencies. The high-speed components will sit idle for long stretches of time while they wait for the slow I/O unit to accept or deliver the desired character. It would be like talking at the normal human rate of 240 words per minute (4 words per second) to someone who could respond only at the rate of 1 word every 8 hours—a difference of 5 orders of magnitude. You wouldn't get much useful work done!

One solution to this problem is to use a device called an **I/O controller**. An I/O controller is like a special-purpose computer whose responsibility is to handle the details of input/output and to compensate for any speed differences between I/O devices and other parts of the computer. It has a small amount of memory, called an *I/O buffer*, and enough *I/O control and logic* processing capability to handle the mechanical functions of the I/O device, such as the read/write head, paper feed mechanism, and screen display. It is also able to transmit to the processor a special hardware signal, called an **interrupt signal**, when an I/O operation is done. The organization of a typical I/O controller is shown in Figure 5.15.

Figure 5.15Organization of an I/O controller

Practice Problems

Assume a disk with the following characteristics:

(called a *double-sided* disk)

to move 1 track in any direction

How many characters can be stored on this disk?

Answer

The total number of characters (ch) is

2 surfaces/disk × 50 tracks/surface × 20 sectors/track × 1024 ch/sector, which is 2,048,000 characters on a single disk.

What are the best-case, worst-case, and average-case access times for this disk? (Assume that the average seek operation must move 20 tracks.)

Answer

The seek time depends on the number of tracks over which the read head must move. This could range from 0, if the arm does not need to move, to a worst case of the arm having to move from the far inside track to the far outside track, a total of 49 tracks. The average, as stated in the problem, is a move across 20 tracks. The best-case rotational delay is 0, whereas the worst case is one complete revolution. The rotational speed is . On the average, we will wait about one-half of a revolution. Finally, the transfer time is the same in all cases, the time it takes for one sector (1/20 of a track) to rotate under the read/write head, which is . Putting all this together in a table produces the following values for the time (in msec) required for each task:
What would be the average-case access time if we could increase the rotation speed from 2,400 rev/min to 7,200 rev/min?

Answer

The new rotational speed is . The seek time is unaffected by the rotational speed. The new average latency time is . The new transfer time is . The .
What would be the average-case access time of the disk of the previous problem if we could reduce the arm movement time to 0.2 msec to move 1 track in any direction? (Again, assume that the average seek operation must move 20 tracks.)

Answer

So the
Defragmenting a disk means to reorganize files on the disk so that as many pieces of the file as possible are stored in sectors on the same track, regardless of the surface it is on. Explain why defragmentation can be beneficial.

Answer

If many pieces of the file are on the same track, then no movement of the read head arm is required and seek time is 0.

What would be the best-case, worst-case, and average-case access time of the disk in Problem 2 if we could reduce the seek time to 0.0? We could do this by having a separate read/write head for every track so there would be no arm motion. This type of "head per track" storage unit is often called a *drum*.

Answer

> Let's assume that we want to display a 12-megapixel photograph on our tablet screen. First, the 12 million bits (assuming 1 bit per pixel; there may be more) representing the photograph are transferred from their current location in primary memory to the I/O buffer storage within the I/O controller. This operation takes place at the high-speed data transfer rates of most computer components—billions or tens of billions of bits per second. Once this information is in the I/O buffer, the processor can instruct the I/O controller to begin the output operation. The control logic of the I/O controller handles the actual transfer and display of these 12 million pixels to the screen. This transfer may be at a much slower rate—perhaps only hundreds or thousands or a few million bits per second. However, the processor does not have to sit idle for the one or two seconds it

may take to display the image. It is free to do something else, perhaps work on another program. The potential slowness of the I/O operation now affects *only* the I/O controller. When all pixels have been displayed, the I/O controller sends an interrupt signal to the processor. The appearance of this special signal indicates to the processor that the I/O operation is finished.

5.2.3 The Arithmetic/Logic Unit

The **arithmetic/logic unit (ALU)** is the subsystem that performs such mathematical and logical operations as addition, subtraction, and comparison for equality. Although they can be conceptually viewed as separate components, in all modern machines the ALU and the control unit (discussed in the next section) have become fully integrated into a single component called the **processor** (the CPU). (Today, virtually all computers contain multiple processing elements called multicore CPUs. We will talk about this type of architecture in Section 5.4.) However, for reasons of clarity and convenience, we will describe the functions of the ALU and the control unit separately.

The ALU is made up of three parts: the registers, the interconnections between components, and the ALU circuitry. Together these components are called the **data path**.

A **register** is a special high-speed storage cell that holds the operands of an arithmetic operation and that, when the operation is complete, holds its result. Registers are quite similar to the random access memory cells described in the previous section, with the following minor differences:

- They do not have a numeric memory address but are accessed by a special *register designator* such as A, X, or R0.
- They can be accessed much more quickly than regular memory cells. Because there are only a few registers (typically, a few dozen up to a few hundred), it is reasonable to utilize the expensive circuitry needed to make the fetch and store operations 5–10 times faster than regular memory cells, of which there will be billions.
- They are not used for general-purpose storage but for specific purposes such as holding operands for an upcoming arithmetic computation.
  For example, an ALU might have three special registers called A, B, and C. Registers A and B hold the two input operands, and register C holds the result. This organization is diagrammed in Figure 5.16.

Figure 5.16 Three-register ALU organization

In most cases, however, three registers are not nearly enough to hold all the values that we might need. A typical ALU will have somewhere between 16 and 128 registers. To see why this many ALU registers are needed, let's take a look at what happens during the evaluation of the expression $(a / b) \times (c - d)$. After we compute the expression (a / b), it would be nice to keep this result temporarily in a high-speed ALU register while evaluating the second expression $(c - d)$. Of course, we could always store the result of (a / b) in a memory cell, but keeping it in a register allows the computer to fetch it more

quickly when it is ready to complete the computation. In general, the more registers available in the ALU, the faster programs will run.

A more typical ALU organization is illustrated in Figure 5.17, which shows an ALU data path containing 16 registers designated R0–R15. Any of the 16 ALU registers in Figure 5.17 could be used to hold the operands of the computation, and any register could be used to store the result.

Figure 5.17 Multiregister ALU organization

To perform an arithmetic operation with the ALU of Figure 5.17, we first move the operands from memory to the ALU registers. Then we specify which register holds the left operand by connecting that register to the communication path called "Left." In computer science terminology, a path for electrical signals is termed a **bus**. We then specify which register to use for the right operand by connecting it to the bus labeled "Right." (Like RAM, registers also use nondestructive fetch so that when it is needed, the value is only copied to the ALU. It is still in the register.) The ALU is enabled to perform the desired operation, and the answer is sent to any of the 16 registers along the bus labeled "Result." (The destructive store principle says that the previous contents of the destination register will be lost.) If desired, the result can be moved from an ALU register back into memory for longer-term storage.

The final component of the ALU is the ALU circuitry itself. These are the circuits that carry out such operations as

$a + b$ (Figure 4.32)

(Figure 4.28)
$a - b$

$a \times b$

a / b

$a < b$

$a > b$

a AND b

Chapter 4 showed how circuits for these operations can be constructed from the three basic logic gates AND, OR, and NOT, and it showed the construction of logic circuits to perform the operations $a + b$ and . The primary concern is how to select the desired operation from among all the possibilities for a given ALU. For example, how do we tell an ALU that can perform the preceding eight operations that we want only the results of one operation, say $a - b$?
One possibility is to use the multiplexer control circuit introduced in Chapter 4 and shown in Figure 4.34. Remember that a multiplexer is a circuit with input lines numbered 0 to , N selector lines, and one output line. The selector lines are interpreted

as a single binary number from 0 to , and the input line corresponding to this number has its value placed on the single output line.

Let's imagine for simplicity that we have an ALU that can perform four functions instead of eight. The four functions are $a + b$, $a - b$, , and *a AND b*, and these operations are numbered 0, 1, 2, and 3, respectively (00, 01, 10, and 11 in binary). Finally, let's assume that every time the ALU is enabled and given values for a and b, it automatically performs all four possible operations rather than just the desired one. These four outputs can be input to a multiplexer circuit, as shown in Figure 5.18.

Figure 5.18 Using a multiplexer circuit to select the proper ALU result

Now place on the selector lines the identification number of the operation whose output we want to keep. The result of the desired operation appears on the output line, and the other three answers are discarded. For example, to select the output of the subtraction operation, we input the binary value 01 (decimal 1) on the selector lines. This places the output of the subtraction circuit on the output line of the multiplexer. The outputs of the addition, comparison, and AND circuits are discarded.

Thus, one possible design philosophy for building an ALU is not to have it perform only the correct operation. Instead, it is to have *every* ALU circuit "do its thing" but then keep only the one desired answer.

Putting Figures 5.17 and 5.18 together produces the overall organization of the ALU of the Von Neumann architecture. This model is shown in Figure 5.19.

Figure 5.19 Overall ALU organization

5.2.4 The Control Unit

The most fundamental characteristic of the Von Neumann architecture is the **stored program**—a sequence of machine language instructions stored as binary values in memory. It is the task of the **control unit** to

- (1) *fetch* from memory the next instruction to be executed,
- (2) *decode* it—that is, determine what is to be done, and
- (3) *execute* it by issuing the appropriate command to the ALU, memory, or I/O controllers.

These three steps are repeated over and over until we reach the last instruction in the program, typically something called HALT, STOP, or QUIT.

To understand the behavior of the control unit, we must first investigate the characteristics of machine language instructions.

Machine Language Instructions. The instructions that can be decoded and executed by the control unit of a computer are represented in **machine language**. Instructions in this language are expressed in binary, and a typical format is shown in Figure 5.20.

Figure 5.20 Typical machine language instruction format

The *operation code* field (referred to by the shorthand phrase *op code*) is a unique unsigned integer value assigned to each machine language operation recognized by the hardware. For example, 0 could be an ADD, 1 could be a COMPARE, and so on. If the operation code field contains *k* bits, then the maximum number of unique machine language operation codes is .

The *address field(s)* are the memory addresses of the values on which this operation will work. If our computer has a maximum of memory cells, then each address field must be *N* bits wide to enable us to address every cell because it takes *N* binary digits to represent all addresses in the range 0 to . The number of address fields in an instruction typically ranges from 0 to about 3, depending on what the operation is and how many operands it needs to do its work. For example, an instruction to add the contents of memory cell X to memory cell Y requires at least two addresses, X and Y. It could require three if the result were stored in a location different from either operand. In contrast, an instruction that tests the contents of memory cell X to see whether it is negative needs only a single address field, the location of cell X.

To see what this might produce in terms of machine language instructions, let's see what the following hypothetical instruction would actually look like when stored in memory.

Operation	Meaning
ADD X, Y	Add contents of memory addresses X and Y and put the sum back into memory address Y.

Let's assume that the op code for ADD is a decimal 9, X and Y correspond to memory addresses 99 and 100 (decimal), and the format of instructions is

A decimal 9, in 8-bit binary, is 00001001. Address 99, when converted to an unsigned 16-bit binary value, is 0000000001100011. Address 100 is 1 greater: 0000000001100100. Putting these values together produces the instruction ADD X, Y as it would appear in memory:

This is somewhat cryptic to a person, but is easily understood by a control unit.

The set of all operations that can be executed by a processor is called its **instruction set**, and the choice of exactly which operations to include or exclude from the instruction set is one of the most important and difficult decisions in the design of a new computer. There is no universal agreement on this issue, and the instruction sets of processors from different vendors may be completely different. This is one reason why a smartphone that uses the Apple A10 processor cannot directly execute programs written for a system that contains an Intel Core i7 found in many popular gaming devices. The operation codes and address fields that these two processors can recognize and carry out are different and completely incompatible.

The machine language operations on most machines are quite elementary, and each operation typically performs a very simple task. The power of a processor comes not

from the sophistication of the operations in its instruction set but from the fact that it can execute each instruction very quickly, typically in a few billionths of a second.

One approach to designing instruction sets is to make them as small and as simple as possible, with perhaps as few as 30–50 instructions. Machines with this sort of instruction set are called *reduced instruction set computers* or **RISC machines**. This approach minimizes the amount of hardware circuitry (gates and transistors) needed to build a processor. The extra space on the chip can be used to optimize the speed of the instructions and allow them to execute very quickly. A RISC processor may require more instructions to solve a problem (because the instructions are so simple), but this is compensated for by the fact that each instruction executes much faster so the overall running time is less. The opposite philosophy is to include a much larger number, say 300–500, of very powerful instructions in the instruction set. These types of processors are called *complex instruction set computers*, or **CISC machines**, and they are designed to directly provide a wide range of powerful features so that finished programs for these processors are shorter. Of course, CISC machines are more complex, more expensive, and more difficult to build. As is often the case in life, it turns out that compromise is the best path—most modern processors use a mix of the two design philosophies.

A little later in this chapter, we will present an instruction set for a hypothetical computer to examine how machine language instructions are executed by a control unit. For clarity, we will not show these instructions in binary, as we did earlier. Instead, we will write out the operation code in English (for example, ADD, COMPARE, MOVE) rather than binary; use the capital letters X, Y, and Z to symbolically represent binary memory addresses; and use the letter R to represent an ALU register. Remember, however, that this notation is just for convenience. All machine language instructions are stored internally using binary representation.

Machine language instructions can be grouped into four basic classes called data transfer, arithmetic, compare, and branch.

1. *Data transfer*. These operations move information between or within the different components of the computer—for example:

- Memory cell → ALU register
- ALU register → memory cell
- One memory cell → another memory cell
- One ALU register → another ALU register

All data transfer instructions follow the nondestructive fetch/destructive store principle described earlier. That is, the contents of the *source cell* (where it is now) are never destroyed, only copied. The contents of the *destination cell* (where it is going) are overwritten, and its previous contents are lost.

Examples of data transfer operations include the following:

Operation	Meaning
LOAD X	Load register R with the contents of memory cell X.
STORE X	Store the contents of register R into memory cell X.
MOVE X,Y	Copy the contents of memory cell X into memory cell Y.

2. *Arithmetic*. These operations cause the arithmetic/logic unit to perform a computation. Typically, they include arithmetic operations like +, −, ×, and /, as well as logical operations such as AND, OR, and NOT. Depending on the instruction set, the operands may reside in memory or they may be in an ALU register.

   Possible formats for arithmetic operations include the following examples. (*Note*: The notation CON(X) means the contents of memory address X.)

Operation	**Meaning**
ADD X,Y, Z	Add the contents of memory cell X to the contents of memory cell Y and put the result into memory cell Z. This is called a *three-address instruction*, and it performs the operation .
ADD X,Y	Add the contents of memory cell X to the contents of memory cell Y. Put the result back into memory cell Y. This is called a *two-address instruction*, and it performs the operation .
ADD X	Add the contents of memory cell X to the contents of register R. Put the result back into register R. This is called a *one-address instruction*, and it performs the operation . (Of course, R must be loaded with the proper value before executing the instruction.)

   Other arithmetic operations such as SUBTRACT, MULTIPLY, DIVIDE, AND, and OR would be structured in a similar fashion.

3. *Compare*. These operations compare two values and set an indicator on the basis of the results of the compare. Most Von Neumann machines have a special set of bits inside the processor called *condition codes* (or a special register called a *status register* or *condition register*); these bits are set by the compare operations. For example, assume there are three 1-bit condition codes called GT, EQ, and LT that stand for greater than, equal to, and less than, respectively. The operation

Operation	**Meaning**
COMPARE X,Y	Compare the contents of memory cell X to the contents of memory cell Y and set the condition codes accordingly. Assume memory cells X and Y hold signed integer values.

4. would set these three condition codes in the following way:

Condition	**How the Condition Codes Are Set**
CON (X) > CON (Y)	
CON (X) < CON (Y)	

5. *Branch*. These operations alter the normal sequential flow of control. The normal mode of operation of a Von Neumann machine is *sequential*. After completing the instruction in address *i*, the control unit executes the instruction in address *i* + 1. (*Note*: If each instruction occupies *k* memory cells rather than 1, then after finishing the instruction starting in address *i*, the control unit executes the instruction starting in address *i* + *k*. In the following discussions, we assume for simplicity that each instruction occupies one memory cell.) The *branch instructions* can alter this sequential mode.

Typically, determining whether to branch is based on the current settings of the condition codes. Thus, a branch instruction is almost always preceded by either a compare instruction or some other instruction that sets the condition codes. Typical branch instructions include the following:

Operation	Meaning
JUMP X	Take the next instruction unconditionally from memory cell X.
JUMPGT X	If the GT indicator is a 1, take the next instruction from memory cell X. Otherwise, take the next instruction from the next sequential location.
(JUMPEQ and JUMPLT would work similarly on the other two condition codes.)	
JUMPGE X	If *either* the GT or the EQ indicator is a 1, take the next instruction from memory location X. Otherwise, take the next instruction from the next sequential location.
(JUMPLE and JUMPNEQ would work in a similar fashion.)	
HALT	Stop program execution. Don't go on to the next instruction.

These are some of the typical instructions that a Von Neumann computer can decode and execute. Problem 2 in the Challenge Work at the end of this chapter asks you to investigate the instruction set of a real processor found inside a modern computer and compare it with what we have described here.

The instructions presented here are quite simple and easy to understand. The power of a Von Neumann computer comes not from having thousands of built-in, high-level instructions but from the ability to combine a great number of these rather simple instructions into large, complex programs that can be executed extremely quickly. Figure 5.21 shows examples of how these simple machine language instructions can be combined to carry out some of the high-level algorithmic operations first introduced in Level 1 and shown in Figure 2.9. (The examples assume that the variables *a, b*, and *c* are stored in memory locations 100, 101, and 102, respectively, and that the instructions occupy one cell each and are located in memory locations 50, 51, 52, ...)

Figure 5.21 Examples of simple machine language instruction sequences

Don't worry if these "miniprograms" are a little confusing. We treat the topic of machine language programming in more detail in the next chapter. For now, we simply want you to know what machine language instructions look like so that we can see how to build a control unit to carry out their functions.

Control Unit Registers and Circuits. It is the task of the control unit to fetch and execute instructions of the type shown in Figures 5.20 and 5.21. To accomplish this task, the control unit relies on two special registers called the **program counter (PC)** and the **instruction register (IR)** and on an *instruction decoder circuit*. The organization of these three components is shown in Figure 5.22.

Figure 5.22 Organization of the control unit registers and circuits

Practice Problems

Assume that the variables *a, b, c*, and *d* are stored in memory locations 100, 101, 102, and 103, respectively, and that the constant value +1 is stored in memory location 104. Using any of the sample machine language instructions given in this section, translate the following pseudocode operations into machine language instruction sequences. Have your instruction sequences begin in memory location 50.

Set *a* to the value $b + c + d$

Answer

Assuming that variables *a*, *b*, *c*, and *d* are stored in memory locations 100, 101, 102, and 103, respectively:

There are many other possible solutions to the previous and the following problems, depending on which instructions you choose to use. The previous solution uses the one-address format. The two- and three-address formats would lead to different sequences.

Set *a* to the value $(b \times d) - (c / d)$

Answer

Set *a* to the value $(a - 1)$

Answer

If then set *c* to the value of *d*
Answer

If $(a \leq b)$ then

Answer

Initialize *a* to the value *d*

Answer

The program counter holds the address of the *next* instruction to be executed. It is like a "pointer" specifying which address in memory the control unit must go to in order to get the next instruction. To get that instruction, the control unit sends the contents of the PC to the MAR in memory and executes the Fetch(address) operation described in Section 5.2.1. For example, if the PC holds the value 73 (in binary, of course), then when the current instruction is finished, the control unit sends the value 73 to the MAR and fetches the instruction contained in cell 73. The PC gets incremented by 1 after each fetch because the normal mode of execution in a Von Neumann machine is sequential. (Again, we are assuming that each instruction occupies one cell. If an instruction occupied *k* cells, then the PC would be incremented by *k*.) Therefore, the PC frequently has its own incrementor (+1) circuit to allow this operation to be done quickly and efficiently.

The instruction register (IR) holds a copy of the instruction just fetched from memory. The IR holds both the op code portion of the instruction, abbreviated , and the address(es), abbreviated .

To determine what instruction is in the IR, the op code portion of the IR must be decoded using an *instruction decoder*. This is the same type of decoder circuit discussed in Section 4.5and used in the construction of the memory unit (Figure 5.8). The *k* bits of the op code field of the IR are sent to the instruction decoder, which interprets them as a numerical value between 0 and . Exactly one of the output lines of the decoder is set to a 1—specifically, the output line whose identification number matches the operation code of this instruction.

Figure 5.23 shows a decoder that accepts a 3-bit op code field and has output lines, one for each of the eight possible machine language operations. The 3 bits of the are fed into the instruction decoder, and they are interpreted as a value from 000 (0) to 111 (7). If the bits are, for example, 000, then line 000 in Figure 5.23 is set to a 1. This line enables the ADD operation because the operation code for ADD is 000. When a 1 appears on this line, the ADD operation

- (1)

  fetches the two operands of the add and sends them to the ALU;

- (2)

  has the ALU perform all of its possible operations;

- (3)

  selects the output of the adder circuit, discarding all others; and

- (4)

  moves the result of the add to the correct location.

Figure 5.23The instruction decoder

If the op code bits are 001 instead, then line 001 in Figure 5.23 is set to a 1. This time the LOAD operation is enabled because the operation code for LOAD is the binary value 001. Instead of performing the previous four steps, the hardware carries out the LOAD operation by:

- (1)sending the value of to the MAR in the memory unit,
- (2)fetching the contents of that address and putting them in the MDR, and
- (3)copying the contents of the MDR into ALU register R.

For every one of the machine language operations in our instruction set, there exists the circuitry needed to carry out, step-by-step, function of that operation. The instruction decoder has output lines, and each output line enables the circuitry that performs the desired operation.

5.3Putting the Pieces Together—the Von Neumann Architecture

We have now described each of the four components that make up the Von Neumann architecture:

- Memory (Figure 5.8)
- Input/output (Figure 5.15)
- ALU (Figure 5.19)
- Control unit (Figures 5.22 and 5.23)

This section puts these pieces together and shows how the entire model functions. The overall organization of a Von Neumann computer is shown in Figure 5.24. Although highly idealized and simplified, the structure in this diagram is quite similar to virtually every computer, tablet, and smartphone ever built!

Figure 5.24 The organization of a Von Neumann computer

To see how the Von Neumann machine of Figure 5.24 executes instructions, let's pick a hypothetical instruction set for our system, as shown in Figure 5.25. We will use the same instruction set in the Laboratory Experience for this chapter and again in Chapter 6 when we introduce and study assembly languages. (*Reminder*: CON(X) means the contents of memory cell X; R stands for an ALU register; and GT, EQ, and LT are condition codes that have the value of 1 for ON and 0 for OFF.)

Figure 5.25

Instruction set for our Von Neumann machine

Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then else 0 if then else 0 if CON(X) < R then else 0

Binary Op Code	Operation	Meaning
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if .
1010	JUMPEQ X	Get the next instruction from memory location X if .
1011	JUMPLT X	Get the next instruction from memory location X if .
1100	JUMPNEQ X	Get the next instruction from memory location X if .
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

The execution of a program on the computer shown in Figure 5.24 proceeds in three distinct phases: *fetch, decode*, and *execute*. These three steps are repeated for every instruction, and they continue until either the computer executes a HALT instruction or there is a fatal error that prevents it from continuing (such as an illegal op code, a nonexistent memory address, or division by zero). Algorithmically, the process can be described as follows:

This repetition of the fetch/decode/execute phase is called the *Von Neumann cycle*. To describe the behavior of our computer during each of these three phases, we will use the following notational conventions:

CON(A) The contents of memory cell A. We assume that an instruction occupies 1 cell.

A → B Send the value stored in register A to register B. The following abbreviations refer to the special registers and functional units of the Von Neumann architecture introduced in this chapter:

PC	The program counter
MAR	The memory address register
MDR	The memory data register
IR	The instruction register, which is further divided into and
ALU	The arithmetic/logic unit

R	Any ALU register
GT, EQ, LT	The condition codes of the ALU
+1	A special increment unit attached to the PC

FETCH Initiate a memory fetch operation (that is, send an F signal on the F/S control line of Figure 5.24).

STORE Initiate a memory store operation (that is, send an S signal on the F/S control line of Figure 5.24).

ADD Instruct the ALU to select the output of the adder circuit (that is, place the code for ADD on the ALU selector lines shown in Figure 5.24).

SUBTRACT Instruct the ALU to select the output of the subtract circuit (that is, place the code for SUBTRACT on the ALU selector lines shown in Figure 5.24).

1. *Fetch phase*. During the fetch phase, the control unit gets the next instruction from memory and moves it into the IR. The fetch phase is the same for every instruction and consists of the following four steps.

   1. PC → MAR Send the address in the PC to the MAR register.
   2. FETCH Initiate a fetch operation using the address in the MAR. The contents of that cell are placed in the MDR.
   3. MDR → IR Move the instruction in the MDR to the instruction register so that we are ready to decode it during the next phase.
   4. PC + 1 → PC Send the contents of the PC to the incrementor and put it back. This points the PC to the next instruction.

2. The control unit now has the current instruction in the IR and has updated the program counter so that it will correctly fetch the next instruction when the execution of this instruction is completed. It is ready to begin decoding and executing the current instruction.

3. *Decode phase*. Decoding the instruction is simple because all that needs to be done is to send the op code portion of the IR to the instruction decoder, which determines its type. The op code is the 4-bit binary value in the first column of Figure 5.25.

   1. → instruction decoder
   The instruction decoder generates the proper control signals to activate the circuitry to carry out the instruction.

4. *Execution phase*. The specific actions that occur during the execution phase are different for each instruction. Therefore, there will be a unique set of circuitry for each of the distinct instructions in the instruction set. The control unit circuitry generates the necessary sequence of control signals and data transfer signals to the other units (ALU, memory, and I/O) to carry out the intent of this instruction. The following examples show what signals and transfers take place during the execution phase of some of the instructions in Figure 5.25 using the Von Neumann model of Figure 5.24.

   a) LOAD X Meaning: Load register R with the current contents of memory cell X.
      1. Send address X (currently in) to the MAR.

	2. FETCH	Fetch contents of cell X and place that value in the MDR.
	3. MDR → R	Copy the contents of the MDR into register R.
b)	STORE X	Meaning: Store the current contents of register R into memory cell X.
	1.	Send address X (currently in) to the MAR.
	2. R → MDR	Send the contents of register R to the MDR.
	3. STORE	Store the value in the MDR into memory cell X.
c)	ADD X	Meaning: Add the contents of cell X to the contents of register R and put the result back into register R.
	1.	Send address X (currently in) to the MAR.
	2. FETCH	Fetch the contents of cell X and place it in the MDR.
	3. MDR → ALU	Send the two operands of the ADD to the ALU.
	4. R → ALU	
	5. ADD	Activate the ALU and select the output of the ADD circuit as the desired result.
	6. ALU → R	Copy the selected result into the R register.
d)	JUMP X	Meaning: Jump to the instruction located in memory location X.
	1.	Send address X to the PC so the instruction stored there is fetched during the next fetch phase.
e)	COMPARE X	Meaning: Determine whether CON(X) < R, , or CON(X) > R, and set the condition codes GT, EQ, and LT to appropriate values. (Assume all codes are initially 0.)
	1.	Send address X to the MAR.
	2. FETCH	Fetch the contents of cell X and place it in the MDR.
	3. MDR → ALU	Send the contents of address X and register R to the ALU.
	4. R → ALU	
	5. SUBTRACT	Evaluate CON(X) – R. The result is not saved, and is used only to set the condition codes. If CON(X) – R > 0, then CON(X) > R and set GT to 1. If , then they are equal and set EQ to 1. If CON(X) – R < 0, then CON(X) < R and set LT to 1.
f)	JUMPGT X	Meaning: If GT condition code is 1, jump to the instruction in location X. We do this by loading the address field of the IR, which is the address of location X, into the PC. Otherwise, continue to the next instruction.
	1. IF THEN	

These are six examples of the sequence of signals and transfers that occur during the execution phase of the fetch/decode/execute cycle. There is a unique sequence of actions for each of the 16 instructions in the sample instruction set of Figure 5.25 and for the approximately 50–300 instructions in the instruction set of a typical Von Neumann computer. When the execution of one instruction is done, the control unit fetches the next instruction, starting the cycle all over again. That is the fundamental sequential behavior of the Von Neumann architecture.

These six examples clearly illustrate the concept of abstraction at work. In Chapter 4, we built complex arithmetic/logic circuits to do operations like addition and comparison. Using these circuits, this chapter describes a computer that can execute machine language instructions such as ADD X and COMPARE X,Y. A machine language instruction such as ADD X is a complicated concept, but it is quite a bit easier to understand than the enormously detailed full adder circuit shown in Figure 4.32, which contains 800 gates and more than 2,000 transistors.

Abstraction has allowed us to replace a complex sequence of gate-level manipulations with the single machine language command ADD, which does addition without our having to know how—the very essence of abstraction. Well, why should we stop here? Machine language commands, although better than hardware, are hardly user friendly.

(Some might even call them “user intimidating.”) Programming in binary and writing sequences of instructions such as

01011010000111110100100001

is cumbersome, confusing, and very error-prone. Why not take these machine language instructions and make them more user oriented and user friendly?

An Alphabet Soup of Speed Measures: MHz, GHz, MIPS, and GFLOPS

It is easy to identify the fastest car, plane, or train—just compare their top speeds in miles per hour (or kilometers per hour) and pick the greatest. However, in the computer arena things are not so simple, and there are many different measures of speed.

The unit you might be most familiar with is *clock speed*, measured in billions of cycles per second, called gigahertz (GHz). The actions of every computer are controlled by a central clock, and the “tick” rate of this clock is one possible speed measure. Processors today have clock rates of about 3–5 GHz, while the fastest processor on the market has a clock rate of 8.80 GHz. However, clock speed can be misleading because a machine’s capability depends not only on the tick rate but also on how much work it can do during each tick. If machine A has a clock rate twice as fast as machine B, but each instruction on machine A takes twice as many clock cycles as machine B to complete, then there is no discernible speed difference.

Therefore, a more accurate measure of machine speed is *instruction rate*, measured in MIPS, an acronym for millions of instructions per second, or GIPS, billions of instructions per second. The instruction rate measures how many machine language instructions of the type listed in Figure 5.25 (e.g., LOAD, STORE, COMPARE, ADD) can be fetched, decoded, and executed in one second. If a computer completes one instruction for every clock cycle, then the instruction rate is identical to the clock rate. However, many instructions require multiple clock ticks, whereas parallel computers can often complete multiple instructions in a single tick. Thus, MIPS and GIPS are a better measure of performance because they can tell you how much work is actually being done, in terms of completed instructions, in a given amount of time.

Finally, some people are only interested in how fast a computer executes the subset of instructions most important to their applications. For example, scientific programs do an enormous amount of floating-point (i.e., decimal) arithmetic, so the computers that execute these programs must be able to execute arithmetic instructions as fast as possible. For these machines, a better measure of speed might be the *floating-point instruction rate*, measured in GFLOPS—billions of floating-point operations per second, TFLOPS, trillions of floating point operations per second, or PFLOPS, quadrillions of floating point operations per second. These are like MIPS and GIPS, except the instructions we focus most closely on are those for adding, subtracting, multiplying, and dividing real numbers.

So, as you can see, there is no universally agreed upon measure of computer speed, and that is what allows different computer vendors all to stand up and claim, “My machine is the fastest!”

Why not give them features that allow us to write correct, reliable, and efficient programs more easily? Why not develop *user-oriented programming languages* designed for people, not machines? This is the next level of abstraction in our hierarchy, and we introduce that important concept in Level 3 of the text.

5.4 Non–Von Neumann Architectures

The Von Neumann architecture, which is the central theme of this chapter, has served the field well for over 60 years, but some computer scientists believe it may be reaching the end of its useful life.

The problems that computers are being asked to solve have grown significantly in size and complexity since the appearance of the first-generation machines in the late 1940s and early 1950s. Designers have been able to keep up with these larger and larger problems by building faster and faster Von Neumann machines. Through advances in hardware design, manufacturing methods, and circuit technology, computer designers have been able to take the basic sequential architecture described by Von Neumann in 1946 and improve its performance by four or five orders of magnitude. First-generation machines were able to execute about 10,000 machine language instructions per second. By the second generation, that had grown to about 1 million instructions per second. Today, even a small desktop PC can perform 1 billion instructions per second, whereas larger and more powerful workstations can execute instructions at the rate of 10–20 billion instructions per second. Figure 5.26 shows the changes in computer speeds from the mid-1940s to the present.

Figure 5.26 Graph of processor speeds, 1945 to the present

(*Note*: The graph shown in Figure 5.26 is logarithmic. Each unit on the vertical axis is 10 times the previous one.) The period from about 1945 to about 1970 is characterized by exponential increases in computation speed. However, as Figure 5.26 shows, even though computer speeds are still increasing, the rate of improvement appears to be slowing down.

This slowdown is due to many things. One important limit on increased processor speed is the inability to place gates any closer together on a chip. (See the Special Interest Box "Moore's Law and the Limits of Chip Design" in Chapter 4.) Today's high-density chips contain billions of transistors separated by distances of only 10–25 nanometers , and it is becoming exceedingly difficult (not to mention expensive) to accurately place individual components closer together. However, the time it takes to send signals between two parts of a computer separated by a given distance is limited by the fact that electronic signals cannot travel faster than the speed of light—299,792,458 meters per second. That is, when we carry out an operation such as:

PC → MAR

the signals traveling between these two registers cannot exceed 300 million meters per second. If, for example, these two components were separated by 1 meter, it would take signals leaving the PC about 3 nanoseconds to reach the MAR, and nothing in this universe can reduce that value except a reduction of the distance separating them.

Even while the rate of increase in the performance of newer machines is slowing down, the problems that researchers are attempting to solve are growing ever larger and more complex. New applications in areas such as computational modeling and high-resolution real-time imaging are increasing the demands placed on new computer systems. For

example, to have a computer generate and display animated images without flicker, it must generate 30 new frames each second. Each frame may contain as many as 4,000 × 4,000 separate picture elements (pixels) whose position, color, and intensity must be individually recomputed. This means that pixel computations need to be completed every second. Each of those computations may require the execution of many instructions. (Where does this point move to in the next frame? What color is it? How bright is it? Is it visible or hidden behind something else?) If we assume that it requires 2,000 instructions per pixel to answer these questions (a reasonable approximation), then real-time computer animation requires a computer capable of executing billion instructions per second. This is well beyond the abilities of most current processors, which are limited to about 20–50 billion instructions per second. The inability of the sequential one-instruction-at-a-time Von Neumann model to handle today's large-scale problems is called the **Von Neumann bottleneck**, and it is a major problem in computer organization.

To solve this problem, computer engineers are rethinking many of the fundamental ideas presented in this chapter, and they are studying nontraditional approaches to computer organization called **non-Von Neumann architectures**. They are asking the question, "Is there a different way to design and build computers that can solve problems 10 or 100 or 1,000 times larger than what can be handled by today's computers?" Fortunately, the answer to this question is a resounding, Yes! (This is equivalent to today's automotive engineers who have stopped trying to improve the performance of the traditional gasoline-powered engine but are investigating totally new designs, such as electric cars and hybrids.)

One of the most important areas of research in these non–Von Neumann architectures is based on the following fairly obvious principle:

If you cannot build something to work twice as fast, build it to do two things at once. The results will be identical.

From this truism comes the principle of **parallel processing**—building computers not with one processor, as shown in Figure 5.24, but with hundreds, thousands, or even tens of thousands. If we can keep each processor occupied with meaningful work, then it should be possible to speed up the solution to large problems by one, two, or three orders of magnitude and overcome the Von Neumann bottleneck. For example, in the graphical animation example discussed earlier, we determined that we needed a machine that could execute 960 billion instructions per second, but the processors currently available may only work at a speed of 20 billion instructions per second. However, let's say that we could have 48 (or more) processors all working together on this one problem; then we should (in theory) have a sufficiently powerful system to solve our problem because . This is the idea behind *dual-core* and *quad-core* processors that have two or four independent processors on a single chip. (Note: The A10 processor inside the Apple iPhone 7 is a quad-core system containing four separate processors.) The approach of placing multiple processors on a single chip is fine for a small number of processors, say two, four, or eight. However, we need a completely different approach to build large-scale parallel systems that may contain thousands or tens of thousands of processors. (Amazingly, the Chinese Sunway TaihuLight computer mentioned in the Special Interest Box "Speed to Burn" contains almost 11 million core processors.)

Speed to Burn

The first computer to achieve a speed of 1 million floating-point operations per second, 1 *megaflop*, was the Control Data 6600 in the mid-1960s. The first machine to achieve 1 billion floating-point operations per second, 1 *gigaflop*, was the Cray X-MP in the early 1980s. Today almost all machines, even inexpensive laptops and tablets, can achieve gigaflop speeds. In 1996, the Intel Corporation announced that its ULTRA computer had successfully become the world's first *teraflop* machine. This $55 million computer contained 9,072 Pentium Pro processors, and on December 16, 1996, it achieved a sustained computational speed of 1 trillion operations per second.

On June 9, 2008, another major milestone in computer performance was reached. The Roadrunner massively parallel computer, constructed jointly by Los Alamos National Laboratories and IBM, achieved a sustained computational speed of 1,026 trillion floating-point operations per second, or 1 *petaflop*. To get an idea of how fast that is, consider that if all 6 billion people in the world worked together on a single problem, each person would have to perform 170,000 computations per second to equal the speed of this one machine. The system, which contained 18,000 processors and 98 terabytes of memory, cost about $100 million to design and build. It was used for basic research in astronomy, energy, and human genome science.

As of early 2017, the fastest computer in the world was the Sunway TaihuLight supercomputer at the National Supercomputing Center in Wuxi, Jiangsu, China. The system contains a total of 10,649,600 core processors, bytes of memory, and cost $US273 million to build. In 2016, it achieved a peak computational rate of 93 petaflops (see "The Tortoise and the Hare" Special Interest Box in Chapter 3). Estimates show that the system, when fully operational, should be able to achieve a peak speed of about 125 petaflops. By the time you are reading this, though, there will likely be an even faster computer. That's how fast the field of parallel computation is moving!

Most large-scale parallel processors use an architecture called **MIMD parallel processing**(*m*ultiple *i*nstruction stream/*m*ultiple *d*ata stream), also called **cluster computing**. In MIMD parallelism, a computer system has multiple, independent processors each with its own primary memory unit, and every processor is capable of executing its own separate program in its own private memory at its own rate. This model of parallel processing is diagrammed in Figure 5.27. (Note that each processor in the diagram of Figure 5.27 may itself contain two, four, or eight independent processors.)

Figure 5.27Model of MIMD parallel processing

Each processor/memory pair in Figure 5.27 is a Von Neumann machine of the type described in this chapter. For example, it could be a processor board of the type shown in Figure 5.24. Alternately, it could be a complete computer system, such as a desktop machine in a computer lab or the laptop in your dorm room. Each system is executing its own program in its own local memory at its own rate. However, rather than each having to solve the entire problem by itself, the problem is solved in a parallel fashion by all processors simultaneously. Each of the processors tackles a small part of the overall problem and then communicates its result to the other processors via the *interconnection network*, a communications system that allows processors to exchange messages and data.

A massively parallel processor would be an excellent system to help us speed up the reverse telephone directory lookup problem initially discussed in Chapter 2. In the sequential approach that we described, the single processor doing the work must search

all 350,000,000 entries from beginning to end (or until the desired telephone number is found). The analysis in Chapter 3 showed that using sequential search and a computer that can examine 50,000 numbers per second, it would take, on average, almost one hour to find a particular number—much too long for the typical person to wait.

However, if we were to use 1,000 processors instead of one, the problem is easily solved—just divide the 350,000,000 numbers into 1,000 equal-sized pieces and assign each piece to a different processor. Now each processor searches *in parallel* to see whether the desired telephone number is in its own section. If it finds the number, it broadcasts that information on the interconnection network to the other 999 processors so that they can stop searching. Each processor needs only to look through a list of 350,000 numbers, which is 1/1000th the amount of work it had to do previously. Instead of an average of 3,500 seconds, almost one hour, we now get our answer in 1/1000th the time—about 3.5 seconds. Parallel processing has elegantly solved our problem.

MIMD parallelism is also a scalable architecture. *Scalability* means that, at least theoretically, it is possible to match the number of processors to the size of the problem. For example, if 1,000 processors were insufficient to solve the reverse telephone lookup problem, then 2,000 or 5,000 can be used instead, assuming the interconnection network can provide the necessary level of communications. (Communications can become a serious bottleneck in a parallel system.) In short, the resources applied to a problem can be in direct proportion to the amount of work that needs to be done and the speed with which the answer is needed. Massively parallel MIMD machines containing hundreds of thousands or millions of independent processors have achieved solutions to large problems thousands of times faster than is possible using a single processor. (For an up-to-date listing of the fastest parallel computers, check the home page of *Top500*, a listing of the 500 most powerful computers in the world. Its web address is www.top500.org.)

The multiple processors within a MIMD cluster do not have to be identical or belong to a single administrative organization. Computer scientists realized that it is possible to address and solve massive problems by utilizing the resources of idle computers located around the world, regardless of whom they belong to. This realization led to an exciting new form of large-scale parallelism called **grid computing**. Grid computing acts much like popular sharing applications that allow homeowners to rent out their houses when they are not there or allow car owners to let someone use their vehicles when they would otherwise be sitting idle.

Grid computing enables researchers to easily and transparently access unused computer facilities without regard for their location. For example, one of the most well-known computer sharing projects is BOINC (Berkeley Open Infrastructure for Networked Computing) hosted by the University of California-Berkeley. BOINC allows users around the world to download BOINC software to their computer (Windows, Mac, Linux, or Android) and then donate idle computer time to assist with a range of scientific projects—curing diseases, studying global warming, analyzing pulsars—that are too massive to be attacked by a single machine. Currently, there are an average of over 672,000 computer systems connected to BOINC at any instant in time, providing a total of 18 petaflops of computing power to this "virtual supercomputer environment." You can read about the activities of BOINC (and learn how to become a contributing part of the system) at http://boinc.berkeley.edu/.

The real key to using massively parallel processors is to design solution methods that effectively utilize the large number of available processors. It does no good to have 1,000 processors available if only 1 or 2 are doing useful work while 998 or 999 are sitting idle. That would be like having a large construction crew at a building site, where the roofers, painters, and plumbers are sitting around waiting for one person to put up the walls. The field of **parallel algorithms**, the study of techniques that make efficient use of parallel architectures, is an important branch of research in computer science. Advances in this area will go a long way toward speeding the development and use of large-scale parallel systems of the type shown in Figure 5.27. (Problem 1 in this chapter's Challenge Work asks you to design a parallel addition algorithm.)

To solve the scientific problems of the 21st century, the computers of the 21st century are being organized much more like the massively parallel processing systems of Figure 5.27than like the 70-year-old Von Neumann model of Figure 5.24.

5.5Summary of Level 2

In Chapter 4, we looked at the basic building blocks of computers: binary codes, transistors, gates, and circuits. This chapter examined the standard model for computer design, called the Von Neumann architecture. It also discussed some of the shortcomings of this sequential model of computation and described briefly how these might be overcome by the use of parallel computers or a system of grid computers.

At this point in our hierarchy of abstractions, we have created a fully functional computer system capable of executing an algorithm encoded as sequences of machine language instructions. The only problem is that the machine we have created is enormously difficult to use and about as unfriendly and unforgiving as it could be. It has been designed and engineered from a machine's perspective, not a person's. Sequences of binary encoded machine language instructions such as

1011010000001011

1001101100010111

0000101101011001

give a computer no difficulty, but they cause people to throw up their hands in despair. We need to create a friendlier environment—to make the computer and its hardware resources more accessible. Such an environment would be more conducive to developing correct solutions to problems and satisfying a user's computational needs.

Quantum Computing

One of the most fascinating (and complex) models of non-Von Neumann computing is called **quantum computing**. In a "regular" (i.e., Von Neumann) machine an individual bit of data is always in a well-defined state–either a 0 or a 1. However, quantum computers are built using the quantum mechanical principle called *superposition*, in which a single bit of data, now called a *qubit*, can be either a 0 or a 1 or *both* a 0 and a 1 simultaneously! In theory, this would allow a quantum computer to examine every possible combination of input values in a single step, greatly speeding up the solution to complex problems. For example, 2 qubits could, at the same time, represent all combinations of two binary values: 00, 01, 10, and 11.

In 2012, a company called D-Wave Systems, of British Columbia, Canada, built the first working model of a quantum computer, the D-Wave 1 with 128 qubits. A second prototype, the D-Wave 2 with 512 qubits, was launched in 2013. Soon after that, NASA, Google, the CIA, and a group of American and Canadian universities formed a research consortium to study how this radically new type of computer system could best be used, and with a goal of demonstrating a working quantum computer by 2020.

If you are a bit confused by the description of this strange new model of computation, don't worry—you have plenty of company. In a February 2014, *Time* magazine story about quantum computing, the following quote appeared on the cover: "It promises to solve some of humanity's most complex problems. It's backed by Jeff Bezos, NASA, and the CIA. Each one costs $10,000,000 and operates at below zero. And nobody knows how it actually works."*

The component that creates this kind of friendly, problem-solving environment is called *system software*. It is an intermediary between the user and the hardware components of the Von Neumann machine. Without it, a Von Neumann machine would be virtually unusable by anyone but the most technically knowledgeable computer experts. We examine system software in the next level of our investigation of computer science.