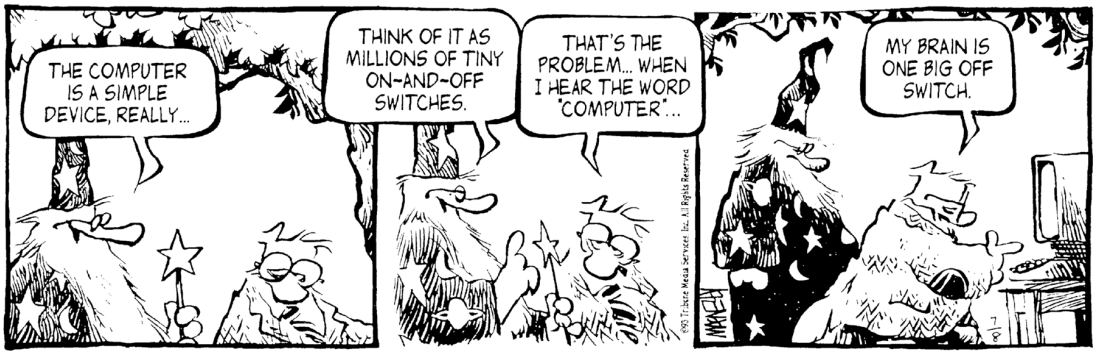
CHAPTER 7

THE CPU AND MEMORY



SHOE © MACNELLY. DISTRIBUTED BY KING FEATURES SYNDICATE

7.0 INTRODUCTION

The previous chapter provided a detailed introduction to the Little Man model of a computer. In that chapter we introduced a format, using a three-digit number divided into op code and address fields, for the instructions that a computer can perform. We introduced an instruction set that we indicated was representative of those found in a real computer. We also showed the steps that are performed by the Little Man in order to execute one of these instructions.

In this chapter and the next we will extend these concepts to the real computer. Our primary emphasis in this chapter is on the central processing unit (CPU), together with memory. In the real computer, memory is actually separated both physically and functionally from the CPU. Memory and the CPU are intimately related in the operation of the computer, however, and so we will treat memory together with the CPU for the convenience of our discussion. Since every instruction requires memory access,[1] it makes sense to discuss the two together.

We will use the Little Man model and its instruction set as a guideline for our discussion. The Little Man instruction set is fundamentally similar to the instruction sets of many different computers. Of course, the Little Man instruction set is based on a decimal number system, and the real CPU is binary, but this is a detail that won't concern us for most of this discussion. The CPU architectural model that we shall discuss is not based on a particular make and model, but is typical of most computers. Chapter 8 will discuss the implementation of this model in modern technology. In Supplementary Chapter 2, we shall look specifically at several popular computer models.

In this chapter you will see that the execution of instructions in the CPU together with memory is nearly identical functionally to the Little Man Computer. There is a one-to-one relationship between the various contents of the mailroom and the functional components of the CPU plus memory. The major differences occur in the facts that the CPU instruction set is created using binary numbers rather than decimal and that the instructions are performed in a simple electronic way using logic based upon Boolean algebra instead of having a Little Man running around a mailroom.

Sections 7.1 through 7.3 present a systematic introduction to the components of the CPU and memory, offering a direct comparison with the components of the Little Man Computer, and focusing on the concept of the register as a fundamental element of CPU operation. In Section 7.4, we show how simple CPU and memory register operations serve as the basic mechanism to implement the real computer's instruction set.

In Section 7.5, we turn our attention to the third major computer system component, the bus component. Buses provide the interconnection between various internal parts of the CPU, and between the CPU and memory, as well as providing

[1] Recall that in the LMC every instruction must be fetched from a mailbox to be executed. The same is true in the real computer.

connections between input and output devices, the CPU, and memory. There are many different types of buses in a computer system, each optimized for a different type of task. Buses can connect two components in a point-to-point configuration or may interconnect several modules in a multipoint configuration. In general, the lines on buses carry signals that represent data, addresses, and control functions. We consider the general requirements for a bus, the features, advantages and disadvantages of different types of buses. In Chapter 11, we will focus on the specific buses that interconnect the various components of a computer system, and show you the ways in which the buses connect different parts of an entire computer system together.

In Sections 7.6, 7.7, and 7.8, we return our attention to the CPU to discuss the characteristics and features of the instruction sets provided in real computers: the different types of instructions, the formats of instruction words, and the general requirements and restraints that are required for instruction words.

You already understand from Chapter 6 how simple instructions can be combined to form the programs that you write. When you complete this chapter, you will have a good understanding of how those instructions are executed in a computer.

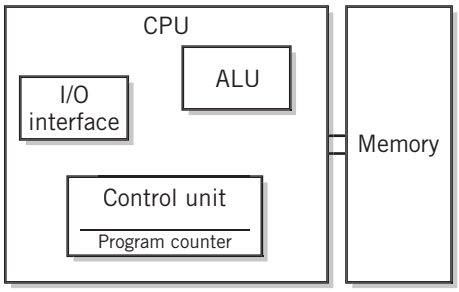
7.1 THE COMPONENTS OF THE CPU

A simplified conceptual block diagram of a CPU with memory is shown in Figure 7.1.[2] For comparison purposes, the block diagram for the Little Man Computer is repeated in Figure 7.2, with labels corresponding to the components in Figure 7.1.

Note the similarities between the two figures. As noted in Chapter 1, the computer unit is made up conceptually of three major components, the **arithmetic/logic unit** (**ALU**), the **control unit** (**CU**), and **memory**. The ALU and CU together are known as the **central processing unit** (**CPU**). An input/output (I/O) interface is also included in the diagram. The I/O interface corresponds in function roughly to the input and output baskets, although its implementation and operation differ from that of the Little Man Computer in many respects.

FIGURE 7.1

System Block Diagram



The arithmetic/logic unit is the component of the CPU where data is held temporarily and where calculations take place. It corresponds directly to the calculator in the Little Man Computer.

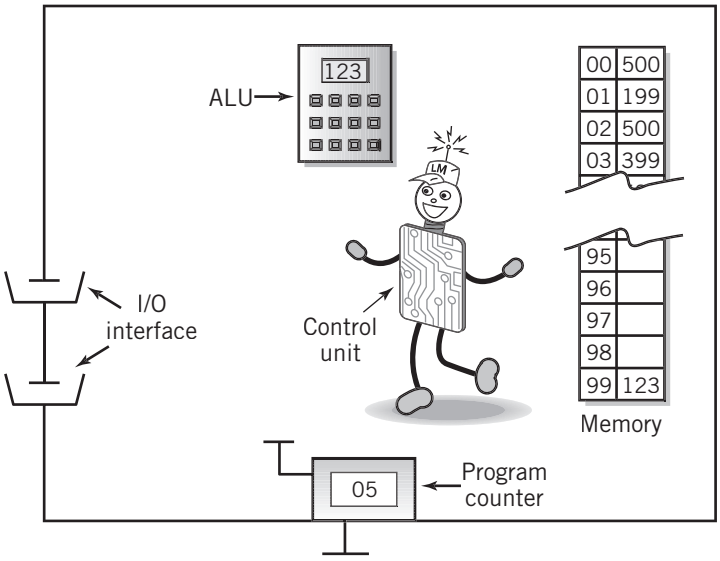
The control unit controls and interprets the execution of instructions. It does so by following a sequence of actions that correspond to the fetch-execute instruction cycle that was described in the previous chapter. Most of these actions are retrievals of instructions from memory followed by movements of data or addresses from one part of the CPU to another.

The control unit determines the particular instruction to be executed by reading the contents of a **program**

[2]This diagram is first attributed to John von Neumann in 1945. As discussed in Chapter 8, current technology results in a different physical layout for the components in the model; nevertheless, the basic execution of instructions is still consistent with the original model.

FIGURE 7.2

The Little Man Computer



counter (**PC**), sometimes called an **instruction pointer**, which is a part of the control unit. Like the Little Man's location counter, the program counter contains the address of the current instruction or the next instruction to be executed. Normally, instructions are executed sequentially. The sequence of instructions is modified by executing instructions that change the contents of the program counter. The Little Man branch instructions are examples of such instructions. A **memory management unit** within the control unit supervises the fetching of instructions and data from memory. The I/O interface is also part of the control unit. In some CPUs, these two functions are combined into a single **bus interface unit**. The program counter in the CPU obviously corresponds to the location counter in the Little Man Computer, and the control unit itself corresponds to the Little Man.

Memory, of course, corresponds directly to the mailboxes in the LMC.

7.2 THE CONCEPT OF REGISTERS

Before we discuss the way in which the CPU executes instructions, it is necessary to understand the concept of a register. A **register** is a single, permanent storage location within the CPU used for a particular, defined purpose. A register is used to hold a binary value temporarily for storage, for manipulation, and/or for simple calculations. Note that each register is wired within the CPU to perform its specific role. That is, unlike memory, where every address is just like every other address, each register serves a particular purpose. The register's size, the way it is wired, and even the operations that take place in the register reflect the specific function that the register performs in the computer.

Registers also differ from memory in that they are not addressed as a memory location would be, but instead are manipulated directly by the control unit during the execution of instructions. Registers may be as small as a single bit or as wide as several bytes, ranging usually from 1 to 128 bits.

Registers are used in many different ways in a computer. Depending on the particular use of a register, a register may hold data being processed, an instruction being executed, a memory or I/O address to be accessed, or even special binary codes used for some other purpose, such as codes that keep track of the status of the computer or the conditions of calculations that may be used for conditional branch instructions. Some registers serve many different purposes, while others are designed to perform a single, specialized task. There are even registers specifically designed to hold a number in floating point format, or a set of related values representing a list or vector, such as multiple pixels in an image.

Registers are basic working components of the CPU. You have already seen, in Chapter 6, that the computer is unable to distinguish between a value that is used as a number in a program and a value that is actually an instruction or address, except in the context of current use. When we refer to the "data" in a register, we might be talking about any of these possibilities.

You have already become acquainted with two "registers" in the Little Man Computer, namely, the calculator and the location counter.

In the CPU, the equivalent to the calculator is known as an **accumulator**. Even the short example to add two numbers in Chapter 6 showed that it is often necessary to move data to and from the accumulator to make room for other data. As a result, modern CPUs provide several accumulators; these are often known as **general-purpose registers**. Some vendors also refer to general-purpose registers as **user-visible** or **program-visible registers** to indicate that they may be accessed by the instructions in user programs. Groups of similar registers are also sometimes referred to collectively as a **register file**. General-purpose registers or accumulators are usually considered to be a part of the arithmetic/logic unit, although some computer manufacturers prefer to consider them as a separate register unit. As in the Little Man Computer, accumulator or general-purpose registers hold the data that are used for arithmetic operations as well as the results. In most computers, these registers are also used to transfer data between different memory locations, and between I/O and memory, again similar to the LMC. As you will see in Chapter 8, they can also be used for some other similar purposes.

The control unit contains several important registers.

- As already noted, the **program counter register** holds the address of the current instruction being executed.
- The **instruction register (IR)** holds the actual instruction being executed currently by the computer. In the Little Man Computer this register was not used; the Little Man himself remembered the instruction he was executing. In a sense, his brain served the function of the instruction register.
- The **memory address register (MAR)** holds the address of a memory location.
- The **memory data register (MDR)**, sometimes known as the *memory buffer register*, will hold a data value that is being stored to or retrieved from the memory location currently addressed by the memory address register.

The last two registers will be discussed in more detail in the next section, when we explain the workings of memory. Although the memory address register and memory data register are part of the CPU, operationally these two registers are more closely associated with memory itself.

The control unit will also contain several 1-bit registers, sometimes known as **flags**, that are used to allow the computer to keep track of special conditions such as arithmetic carry and overflow, power failure, and internal computer error. Usually, several flags are grouped into one or more **status registers**.

In addition, our typical CPU will contain an I/O interface that will handle input and output data as it passes between the CPU and various input and output devices, much like the LMC *in* and *out* baskets. For simplification, we will view the I/O interface as a pair of I/O registers, one to hold an I/O address that addresses a particular I/O device, the other to hold the I/O data. These registers operate similarly to the memory address and data registers. Later, in Chapter 9, we will discuss a more common way of handling I/O that uses memory as an intermediate storage location for I/O data.

Most instructions are executed by the sequenced movement of data between the different registers in the ALU and the CU. Each instruction has its own sequence.

Most registers support four primary types of operations:

1. Registers can be loaded with values from other locations, in particular from other registers or from memory locations. This operation destroys the previous value stored in the destination register, but the source register or memory location remains unchanged.
2. Data from another location can be added to or subtracted from the value previously stored in a register, leaving the sum or difference in the register.
3. Data in a register can be shifted or rotated right or left by one or more bits. This operation is important in the implementation of multiplication and division. The details of the shift operation are discussed in Section 7.6.
4. The value of data in a register can be tested for certain conditions, such as zero, positive, negative, or too large to fit in the register.

In addition, special provision is frequently made to load the value zero into a register, which is known as clearing a register, and also to invert the 0s and 1s (i.e., take the 1's complement of the value) in a register, an operation that is important when working with complementary arithmetic. It is also common to provide for the addition of the value 1 to the value in a register. This capability, which is known as incrementing the register, has many benefits, including the ability to step the program counter, to count in for loops, and to index through arrays in programs. Sometimes decrementing, or subtraction of 1, is also provided. The bit inversion and incrementing operations are combined to form the 2's complement of the value in a register. Most computers provide a specific instruction for this purpose, and also provide instructions for clearing, inverting, incrementing, and decrementing the general-purpose registers.

The control unit sets (''1'') or resets (''0'') status flags as a result of conditions that arise during the execution of instructions.

As an example, Figure 7.3 identifies the programmer-accessible registers in the IBM System z computers, which includes a variety of IBM mainframe models. Internal registers, such as the instruction, memory address, and memory buffer registers are not specifically

FIGURE 7.3

Programmer-Accessible Registers in IBM zSeries Computers

Register type	Number	Size of each in bits	Notes
General	16	64	For arithmetic, logical, and addressing operations; adjoining registers may be joined to form up to eight 128-bit registers
Floating point	16	64	Floating point arithmetic; registers may be joined to form 128-bit registers
PSW	1	128	Combination program counter and status-flag register, called the **Program Status Word (PSW)**
Control (+1 32-bit floating point control)	16	64	Various internal functions and parameters connected with the operating system; accessible only to systems programmers

identified in the table, since they are dependent on the implementation of the particular model in the series.

7.3 THE MEMORY UNIT

The Operation of Memory

To understand the details of instruction execution for the real CPU, you need first to see how instructions and data can be retrieved from memory. Real memory, like the mailboxes in the Little Man Computer, consists of cells, each of which can hold a single value, and each of which has a single address.

Two registers, the memory address register and the memory data register, act as an interface between the CPU and memory. The memory data register is called the memory buffer register by some computer manufacturers.

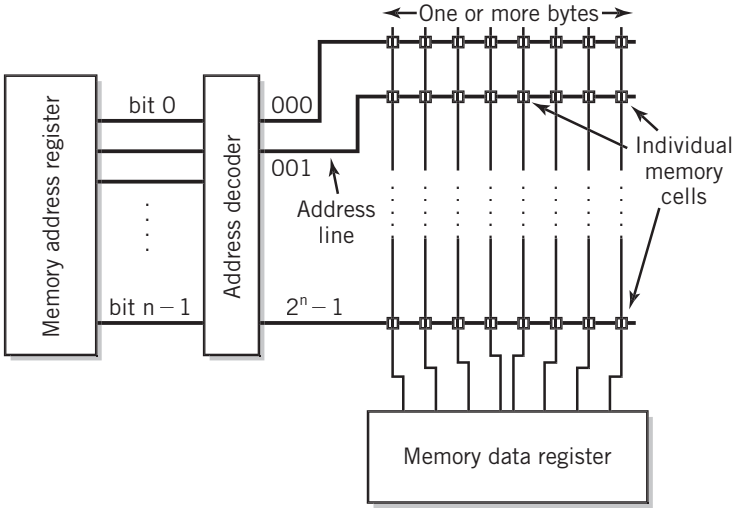
Figure 7.4 is a simplified representation of the relationship between the MAR, the MDR, and memory. Each cell in the memory unit holds 1 bit of data. The cells in Figure 7.4 are organized in rows. Each row consists of a group of one or more bytes. Each group represents the data cells for one or more consecutive memory addresses, shown in the figure as addresses 000, 001, . . . , $2^n - 1$.

In modern computers, it is common to address 8 bytes at a time to speed up memory access between the CPU and memory. The CPU can still isolate individual bytes from the group of eight for its use, however.

The memory address register holds the address in the memory that is to be ''opened'' for data. The MAR is connected to a decoder that interprets the address and activates a single address line into the memory. There is a separate address line for each group of cells in the memory; thus, if there are n bits of addressing, there will be 2^n address lines. (In actuality, the decoding process is somewhat more complex, involving several levels of address decoding, since there may be several millions or billions of addresses involved, but the concept described here is correct.)

FIGURE 7.4

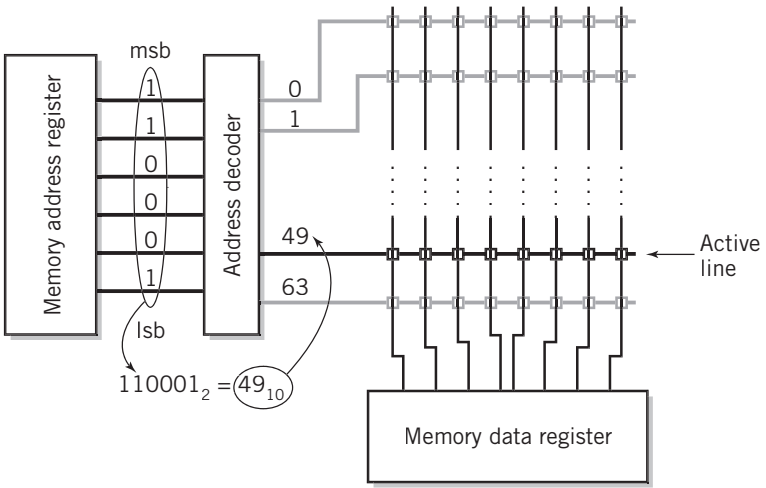
The Relationship Between the MDR, the MAR, and Memory



The memory data register is designed such that it is effectively connected to every cell in the memory unit. Each bit of the MDR is connected in a column to the corresponding bit of every location in memory. However, the addressing method assures that only a single row of cells is activated at any given time. Thus, only one memory location is addressed at any one time. A specific example of this is shown in Figure 7.5. (Note that in the drawing *msb* stands for most significant bit and *lsb* for least significant bit.)

FIGURE 7.5

MAR-MDR Example



As a simple analogy to the operation we've just described, consider the memory as being stored in a glass box, as shown in Figure 7.6. The memory data register has a window into the box. The viewer, who represents each cell in the memory data register, can see the cells in corresponding bit position for every location in memory through the window. The cells themselves are light bulbs that can be turned on (1) or off (0). The output from the memory address register is passed to an address decoder. The output from the address decoder in our analogy consists of a series of lines, each of which can light up the bulbs in a single row of cells. Only one line at a time can be activated—specifically, the one corresponding to the decoded address. The active line will light the bulbs that correspond to "1s," leaving the "0s" dark. The viewer therefore will see only the single group of cells that is currently addressed by the memory address register. We can extend the analogy to include a "master switch" that controls all the lights, so that the data can be read only at the appropriate instant.

A more detailed picture of an individual memory cell is shown in Figure 7.7. Although this diagram is a bit complicated, it may help to clarify how data is transferred between the MDR and memory. There are three lines that control the memory cell: an address line, a read/write line, and an activation line. The address line to a particular cell is turned on only if the computer is addressing the data within that cell. The read/write line determines whether the data will be transferred from the cell to the MDR (read) or from the MDR to the cell (write). This line works by turning on one of two switches in conjunction with the address line and the activation line. The read switch, R, in the diagram turns on when the

FIGURE 7.6

A Visual Analogy for Memory

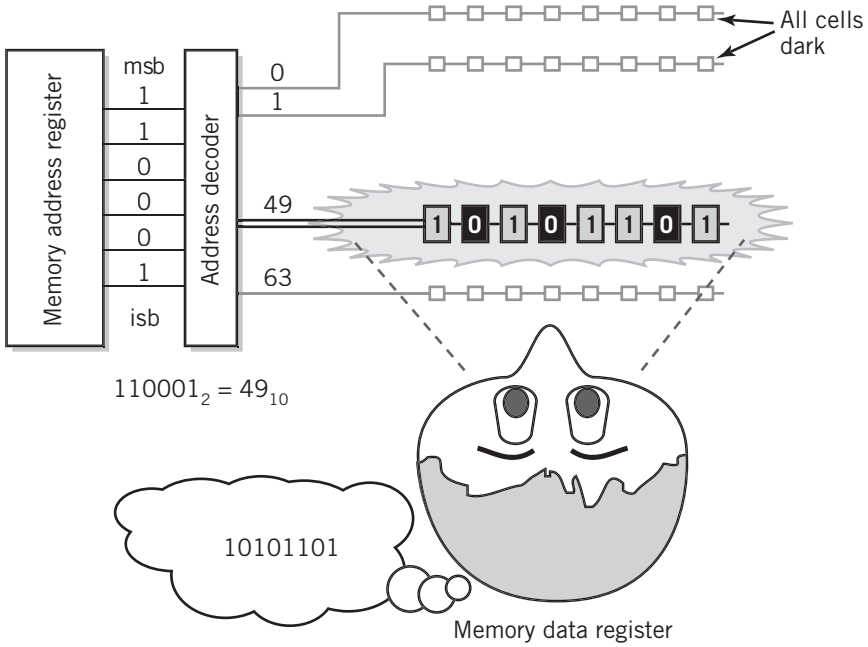
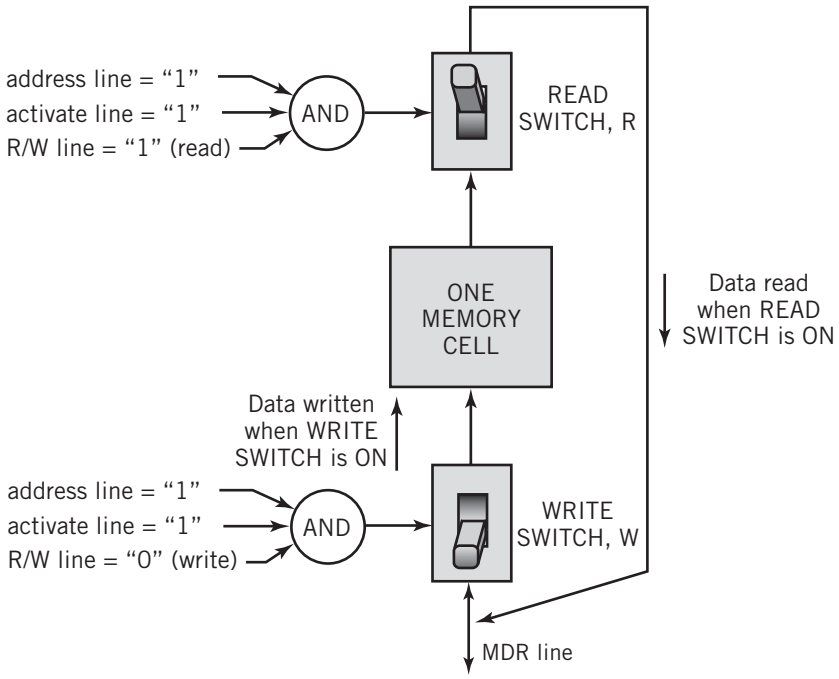


FIGURE 7.7

An Individual Memory Cell



address line and the activation line are both on (on is usually represented by 1, off by 0), and the read/write line is set to read; the switch then connects the output of the cell to the MDR line. The write switch, W, works similarly; switch W turns on when the address line and activation line are both on and the read/write switch is set to write. Switch W connects the MDR line to the input of the cell, which transfers the data bit on the MDR line to the cell for storage. Note that only one switch, at most, can be on at a given time.

The interaction between the CPU and the memory registers takes place as follows: to retrieve or store data at a particular memory location, the CPU copies an address from some register in the CPU to the memory address register. *Note that addresses are always moved to the MAR; there would never be a reason for an address transfer from the MAR to another register within the CPU*, since the CPU controls memory transfers and is obviously aware of the memory address being used. At the same time that the MAR is loaded, the CPU sends a message to the memory unit indicating whether the memory transfer is a retrieval from memory or a store to memory. This message is sent by setting the read/write line appropriately.

At the appropriate instant, the CPU momentarily turns on the switch that connects the MDR with the register by using the activation line, and the transfer takes place between memory and the MDR. The MDR is a two-way register. When the instruction being executed is to store data, the data will be transferred from another register in the CPU to the MDR, and from there it will be transferred into memory. The original data at that

location will be destroyed, replaced by the new data from the MDR. Conversely, when the instruction is to load data from memory, the data is transferred from memory to the MDR, and it will subsequently be transferred to the appropriate register in the CPU. In this case, the memory data are left intact, but the previous data value in the MDR is replaced by the new data from memory.

Memory Capacity

The number of possible memory locations in the Little Man Computer, one hundred locations, was established by the two-digit address space in each instruction. The location counter also addresses one hundred locations. There is no memory address register per se, but the Little Man is certainly aware that each memory location requires two digits. In theory, a larger location counter, say, three digits, would allow the Little Man to fetch more *instructions*, but notice that his *data* fetches and stores are still limited to the one hundred locations that the two digits of the address field in the instruction word can address.

Similarly, there are two factors that determine the capacity of memory in a real computer. The number of bits in the memory address register determines how many different address locations can be decoded, just as the two-digit addresses in the Little Man Computer resulted in a maximum of one hundred mailboxes. For a memory address register of width k bits, the number of possible memory addresses is

$$M = 2^k$$

The other factor in establishing memory capacity is of course the number of bits in the address field of the instruction set, which establishes how many memory locations can be directly addressed from the instruction.

In the Little Man Computer, we have assumed that these two size factors are the same, but in a real computer, that is not necessarily the case. Even if the size of the instruction address field is sufficient to support a larger amount of memory, the number of physical memory locations is, in fact, determined by the size of the memory address register. There are also other ways of extending the addresses specified within instructions so that we can reach more addresses than the size of the instruction address field would allow. Just to give you one common method, consider a computer that can use one of the general-purpose registers to hold an address. To find a memory location, the computer would use the value in that register as a pointer to the address. Instead of an address field, the instruction needs only to indicate which register contains the address. Using this technique, the addressing capability of the computer is determined by the size of the register. For example, a computer with 64-bit registers could address 2^{64} addresses if the MAR were wide enough. Such an extension would suggest that the MAR, and thus the actual memory capacity, is normally at least as large as the instruction address field, but it may be much larger. There is a brief discussion of simple addressing methods in Chapter 8. Additional, more sophisticated addressing methods are presented in Supplementary Chapter 3.

Ultimately, the width of the MAR determines the maximum amount of addressable memory in the computer. Today, a typical memory address register will be at least 32 bits wide, and probably much wider. Many modern CPUs support 64-bit memory addresses. A 32-bit memory address allows a memory capacity of 4 gigabytes (GB) (4×10^9 byte-size spaces), whereas 64 bits allows a memory capacity of 16×10^{18} bytes (16 exabytes or

16 billion gigabytes). In modern computers, the ultimate size of memory is more likely limited by physical space for the memory chips or by the time required to decode and access addresses in a large memory, rather than by the capability of the CPU to address such a large memory.

Of course the size of memory also affects the speed of access. The time needed for the address decoder to identify a single line out of four billion is necessarily larger than that required for a memory that is much smaller.

As an aside, it is worth noting that early models of IBM's largest mainframe computer systems had a total memory capacity of only 512 KB, (1/4000th the memory of a typical modern PC with 2 GB of memory!) and that the original IBM PC came supplied with 64 KB of memory, with a maximum capacity of 640 KB. In fact, Bill Gates, of Microsoft, was quoted at the time as saying that he could see no need for more than 640 KB of memory, ever!

The size of the data word to be retrieved or stored in a single operation is determined by the size of the memory data register and by the width of the connection between memory and the CPU. In most modern computers, data and instructions found in memory are addressed in multiples of 8-bit bytes. This establishes the minimum instruction size as 8 bits. Most instructions cannot fit practically into 8 bits. If one were to allow 3 bits for the op code (eight instruction types), only 5 bits remain for addressing. Five bits allows $2^5 = 32$ different addresses, which is clearly insufficient address space. As a result, longer instructions of 16, 24, 32, or even more bits will be stored in successive memory locations. In the interest of speed, it is generally desirable to retrieve an entire instruction with a single fetch, if possible. Additionally, data to be used in arithmetic calculations frequently requires the precision of several bytes. Therefore, most modern computer memories are designed to allow the retrieval or storage of at least 4 and, more commonly, 8 or even 16, successive bytes in a single operation. Thus, the memory data register is usually designed to retrieve the data or instruction(s) from a sequence of several successive addresses all at once, and the MDR will be several bytes wide.

Primary Memory Characteristics and Implementation

Through the history of computing there have been several different types of primary memory used, reflecting the technology and the system requirements and capabilities of the times. In the 1960s and 1970s, the dominant technology was magnetic core memory, which used a tiny core of magnetic material to hold a bit of data, and the largest machines might have had 512 KB of memory. Today, the primary memory in most computer systems is dynamic **RAM**, and most machines have 1 GB of memory, or more. RAM is an acronym that stands for *random access memory*, which is a slight misnomer, since other types of semiconductor memory can also be accessed randomly (i.e., their addresses can be accessed in any order). A more appropriate name would be *read-write memory*.

Memory today is characterized by two predominant operational factors and by a number of technical considerations. Operationally, the most important memory characteristic is whether the memory is read-write capable or read-only. Almost as important is whether the memory is **volatile** or **nonvolatile**. Nonvolatile memory retains its values when power is removed. Volatile memory loses its contents when power is removed. Magnetic core memory was nonvolatile. RAM is volatile.

Important technical considerations include the speed of memory access, the total amount of memory that can be addressed, the data width, the power consumption and heat generation, and the bit density (specified as the number of bits per square centimeter). Cost is an additional factor.

Most current computers use a mix of static and dynamic RAM for memory. The difference between static and dynamic RAM is in the technical design and is not of importance here. However, **dynamic RAM** is less expensive, requires less electrical power, generates less heat, and can be made smaller, with more bits of storage in a single integrated circuit. Dynamic RAM also requires extra electronic circuitry that ''refreshes'' memory periodically; otherwise the data fades away after a while and is lost. Static RAM does not require refreshing. **Static RAM** is also faster to access than dynamic RAM and is therefore useful in very-high-speed computers and for small amounts of high-speed memory, but static RAM is lower in bit density and more expensive. Both dynamic and static RAM are volatile: their contents are lost when power is turned off.

At the time of this writing, dynamic RAM is standard for most applications. The amount of data that can be stored in a single dynamic RAM chip has increased rapidly in the past few years, going from 64 Kilobits(Kb) to 512 Megabits (Mb) in fewer than fifteen years. Currently, most systems are built with chips that can hold 256 or 512 Mb of data. These chips are also designed to be packaged together in convenient plug-in packages that can supply 1–2 gigabytes of memory, or more, in a single unit. 1 gigabit (Gb) and 2 Gb RAM chips are also in production, but have not yet replaced 512 Mb chips for most system applications, with the exception of mainframe systems. Most modern systems also provide a small amount of static RAM memory that is used for high-speed access. This memory is known as *cache memory*. The use of cache memory is discussed in Chapter 8.

Although current RAM technology is fast, inexpensive, and efficient, its volatility makes some applications difficult or impossible. For example, nonvolatile RAM would make it possible to shut off a computer without losing the programs and data in memory. This would make it possible to restart the computer into its previous state without rebooting, would eliminate the undesirable effects of power failures and laptop battery discharge, and would simplify the use of computers in situations where power conservation is critical, such as in long distance space missions. The desire for nonvolatile RAM has led to considerable research on alternative technologies for creating and producing nonvolatile RAM.

There are a small number of memory technologies in current use that are capable of nonvolatile random access, but none in current large-scale production is capable of replacing standard SRAM and DRAM for use in primary memory. Foremost among these technologies is **flash memory**. Flash memory uses a concept called *hot carrier injection* to store bits of data. Flash memory allows rewriting of cells by erasing groups of memory cells selectively, and then writing the new pattern into the cells. Flash memory serves as an inexpensive form of nonvolatile storage for portable computer storage, digital cameras. MP3 players, and other electronic devices; however it is unsuitable for primary memory because the rewrite time is extremely slow compared to standard RAM and the number of rewrites over the lifetime of the ROM is somewhat limited. Flash memory is viewed primarily as a potential replacement for slow long-term storage devices such as magnetic disks and CD or DVD devices, although the significantly higher cost of flash memory is still a factor at this point in time.

A number of nonvolatile memory technologies that might be capable of replacing traditional RAM appear to be nearing production. These include magnetorestrictive RAM (MRAM), ferroelectric RAM (FeRAM), phase-change RAM (PRAM), and carbon nano[tube] RAM (NRAM). You will probably be reading about one or more of these in the future.

ROM, or *read-only memory*, is used for situations where the software is built semi-permanently into the computer, is required as part of the computer's software, and is not expected to change over the life of the computer, except perhaps very infrequently. Bootstrap programs and basic I/O system drivers fall into this category. Early ROM memory was made up of integrated circuits with fuses in them that could be blown. These fuses were similar to, but much smaller than, the fuses that you might have in your home. A blown fuse might represent a "0," an intact fuse a "1." Modern ROM memories use a different technology, such as **EEPROM** or flash memory. EEPROM (Erasable Electrically Programmable ROM) uses a concept called Fowler-Nordheim tunneling to achieve rewritability. Because of its cost, need for special circuitry, and speed, EEPROM has mostly been replaced by flash memory. Regardless of technology, ROM is nonvolatile. Thus, although electrical power is required to access the data, the data remains consistent with or without power.

7.4 THE FETCH-EXECUTE INSTRUCTION CYCLE

The fetch-execution instruction cycle is *the* basis for every capability of the computer. This seems like a strong statement, but think about it: the purpose of the computer is to execute instructions similar to those that we have already introduced. And, as you've already seen from the Little Man Computer, the operation of every instruction is defined by its fetch-execute instruction cycle. Ultimately, the operation of a computer as a whole is defined by the primary operations that can be performed with registers, as explained in Section 7.2: to move data between registers, to add or subtract data to a register, to shift data within a register, and to test the value in a register for certain conditions, such as negative, positive, or zero.

With the importance of the instruction cycle in mind, we can consider how these few operations can be combined to implement each of the instructions in a computer. The registers that will be of the most importance to us for this discussion will be the general-purpose registers or accumulators used to hold data values between instructions (A or GR), the program counter (PC), which holds the address of the current instruction, the instruction register (IR), which will hold the current instruction while it is being executed, and the memory address and data registers (MAR and MDR), used for accessing memory.

To begin, review carefully the steps that the Little Man took to execute an instruction. (You may want to read Section 6.6 again to refresh your memory.) You will recall that there were two phases in the process. First, the Little Man fetched the instruction from memory and read it. This phase was identical for every instruction. Then, he interpreted the instruction and performed the actions required for that particular instruction.

He repeated this cycle endlessly, until he was given the instruction to stop.

The **fetch-execute instruction cycle** in a CPU works similarly. As noted, much of the procedure consists of copying data from one register to another. You should always be

aware that data copying does not affect the "from" register, but it obviously replaces the previous data in the "to" register with the new data being copied.

Remember that every instruction must be fetched from memory before it can be executed. Therefore, the first step in the instruction cycle always requires that the instruction must be fetched from memory. (Otherwise, how would the computer know what instruction to perform?) Since the address of the current instruction to be executed is identified by the value in the program counter register, the first step will be to transfer that value into the memory address register, so that the computer can retrieve the instruction located at that address.

We will use the following notation to indicate the transfer of a data value from one register to another:

$$REG_a \rightarrow REG_b$$

Then, in this notation, the first step in the execution of every instruction will be

$$\text{(step 1)}\ PC \rightarrow MAR$$

As explained in the description of memory, this will result in the instruction being transferred from the specified memory location to the memory data register. The next step is to transfer that instruction to the instruction register:

$$\text{(step 2)}\ MDR \rightarrow IR$$

The instruction register will hold the instruction through the rest of the instruction cycle. It is the particular instruction in the IR that will control the particular steps that make up the remainder of the cycle. These two steps comprise the fetch phase of the instruction cycle.

The remaining steps are, of course, instruction dependent. Let us consider the steps required to complete a LOAD instruction.

The next thing that the Little Man did was to read the address part of the LOAD instruction. He then walked over to the mailbox specified by that address, read the data, and copied it into the calculator. The real CPU will operate similarly, substituting register transfers for the Little Man, of course. Thus,

$$\text{(step 3)}\ IR[address] \rightarrow MAR$$

The notation IR [address] is used to indicate that only the address part of the contents of the instruction register is to be transferred. This step prepares the memory module to read the actual data that will be copied into the "calculator," which in this case will be the accumulator:

$$\text{(step 4)}\ MDR \rightarrow A$$

The CPU increments the program counter, and the cycle is complete and ready to begin the next instruction (actually this step can be performed any time after the previous instruction is retrieved, and is usually performed early in the cycle in parallel with other steps).

$$\text{(step 5)}\ PC + 1 \rightarrow PC$$

Notice the elegant simplicity of this process! The LOAD instruction requires only five steps. Four of the steps simply involve the movement of data from one register to another.

The fifth step is nearly as simple. It requires the addition of the value 1 to the contents of a register, and the new value is returned to the same register. This type of addition is common in computers. In most cases, the result of an addition or subtraction is returned to one of the original registers.

The remaining instructions operate similarly. Compare, for example, the steps required to perform the STORE and the ADD instructions with those of the LOAD instruction, discussed earlier.

The STORE instruction

```
PC → MAR
MDR → IR
IR[address] → MAR
A → MDR
PC + 1 → PC
```

The ADD instruction

```
PC → MAR
MDR → IR
IR[address] → MAR
A + MDR → A
PC + 1 → PC
```

Study these examples carefully. For practice, relate them to the steps the Little Man performs to execute the corresponding instruction. Notice that the only step that changes in these three instructions is the fourth step.

The fetch-execute cycles for the remaining instructions are left as an exercise (see Exercise 7.5 at the end of this chapter).

The following example, with comments, recaps the above discussion in the context of a three-instruction program segment that loads a number from memory, adds a second number to it, and stores the result back to the first memory location. Note that each instruction is made up of its corresponding fetch-execute cycle. The program segment is executed by processing each step of each fetch-execute cycle in sequence.

Assume that the following values are present just prior to execution of this segment:

```
Program Counter: 65
Value in Mem Location 65: 590 (LOAD 90)
Value in Mem Location 66: 192 (ADD 92)
Value in Mem Location 67: 390 (STORE 90)
Value in Mem Location 90: 111
Value in Mem Location 92: 222
```

EXAMPLE

1st instruction LOAD 90:	PC → MAR	MAR now has 65
	MDR → IR	IR contains the instruction: 590
	- - - - - - - - - - - - - ←	end of fetch
	IR [address] → MAR	MAR now has 90, the location of the data
	MDR → A	Move 111 from MDR to A
	PC + 1 → PC	PC now points to 66.
	———————————————	end of execution, end of first instruction

```
2nd instruction ADD 92:  PC → MAR              MAR now contains 66
                         MDR → IR              IR contains the instructions: 192
                         - - - - - - - - - - - - ← end of fetch
                         IR [address] → MAR    MAR now has 92
                         A + MDR → A           111+222=333 in A
                         PC + 1 → PC           PC now points to 67

————————————————————————————— end of execution, end of second instruction

3rd instruction STORE 90: PC → MAR             MAR now contains 67
                          MDR → IR             IR contains 390
                          - - - - - - - - - - - - ← end of fetch
                          IR [address] → MAR   MAR now holds 90
                          A → MDR              The value in A, 333, moves to mem
                                                 location 90
                          PC + 1 → PC          PC now points to 68

————————————————————————————— end of execution, end of third instruction
←        ready for next instruction
```

7.5 BUSES

Bus Characteristics

You have already seen that instructions are executed within the CPU by moving "data" in many different forms from register to register and between registers and memory. The different forms that the "data" can take include instructions and addresses, in addition to actual numerical data. "Data" moves between the various I/O modules, memory, and the CPU in similar fashion. The physical connection that makes it possible to transfer data from one location in the computer system to another is called a **bus**. From our previous discussion of the way that the CPU and memory work together, it is probably already obvious to you that there must be a bus of some kind linking the CPU and memory; similarly, buses internal to the CPU can be used to link registers together at the proper times to implement the fetch-execute cycles introduced in Section 7.4.

Specifically, a bus may be defined as a group of electrical, or, less commonly, optical, conductors suitable for carrying computer signals from one location to another. The electrical conductors may be wires, or they may be conductors on a printed circuit. Optical conductors work similarly, using light that is directed from point to point in special thin clear glass fibers. Optical conductors can carry data much faster than electrical conductors, but their cost is high, which has limited their use to date. Nonetheless, there is considerable lab research into ways to integrate more optical circuits into computers.

Buses are used most commonly for transferring data between computer peripherals and the CPU, for transferring data between the CPU and memory, and for transferring data between different points within the CPU. A bus might be a tiny fraction of an inch long, carrying data between various parts of the CPU within an integrated circuit chip; it might be a few inches long, carrying data between the CPU chip and memory; it might even be

hundreds of feet long, carrying data between different computers connected together in a network.

The characteristics of buses are dependent on their particular use within the computer environment. A bus can be characterized by the number of separate wires or optical conductors in the bus; by its *throughput*, that is, the data transfer rate measured in bits per second; by the data width (in bits) of the data being carried; by the number and type of attachments that the bus can support; by the distance between the two end points; by the type of control required; by the defined purpose of the bus; by the addressing capacity; by whether the lines on the bus are uniquely defined for a single type of signal or shared; and by the various features and capabilities that the bus provides. The bus must also be specified electrically and mechanically; by the voltages used; by the timing and control signals that the bus provides, by the protocol used to operate and control the bus, by the number of pins on the connectors, if any; even by the size of the cards that plug into the connector. A bus would not be very useful if the cards that it was to interconnect did not fit into the space allotted! Unfortunately for the concept of standardization, there are dozens of different buses in use, although a few are far more common than others.

The need to characterize buses comes from the necessity of interfacing the bus to other components that are part of the computer system. Buses that are internal to the CPU are usually not characterized formally at all, since they serve special purposes and do not interface to the outside world. Buses that are used in this way are sometimes known as *dedicated* buses. Buses that are intended for more general use must have a well-defined standard; standard buses generally have a name. PCI Express, USB, IDE, and SATA are all examples of named buses.

Each conductor in the bus is commonly known as a **line**. Lines on a bus are often assigned names, to make individual lines easier to identify. In the simplest case, each line carries a single electrical signal. The signal might represent one bit of a memory address, or a sequence of data bits, or a timing control that turns a device on and off at the proper time. Sometimes, a conductor in a bus might also be used to carry power to a module. In other cases, a single line might represent some combination of functions.

The lines on a bus can be grouped into as many as four general categories: data, addressing, control, and power. Data lines carry the "data" that is being moved from one location to another. Address lines specify the recipient of data on the bus. Control lines provide control and timing signals for the proper synchronization and operation of the bus and of the modules and other components that are connected to the bus. A bus connecting only two specific 32-bit registers within a CPU, for example, may require just thirty-two data lines plus one control line to turn the bus on at the correct time. A backplane that interconnects a 64-bit data width CPU, a large memory, and many different types of peripherals might require many more than a hundred lines to perform its function.

The bus that connects the CPU and memory, for example, needs address lines to pass the address stored in the MAR to the address decoder in memory and data lines to transfer data between the CPU and the memory MDR. The control lines provide timing signals for the data transfer, define the transfer as a read or write, specify the number of bytes to transfer, and perform many other functions.

In reality, all of the lines except for the power lines in a bus can be used in different ways. Each line in a bus may serve a single, dedicated purpose, such as a bus line that carries the twelfth bit of an address, for example. Alternatively, a line may be configured to serve

different purposes at different times. A single line might be used to carry each of the bits of an address in sequence, followed by the bits of data, for example. At their two extremes, buses are characterized as **parallel** or **serial**. By definition, a parallel bus is simply a bus in which there is an individual line for each bit of data, address, and control being used. This means that all the bits being transferred on the bus can be transferred simultaneously. A serial bus is a bus in which data is transferred sequentially, one bit at a time, using a single data line pair. (A data return line is required to complete the circuit, just as there are two wires in a standard 110-volt power circuit. Multiple data lines can share the same data return line, commonly known as a *ground* line, but in some cases it is possible to reduce noise and other interference by using a separate return line for each data line.)

A bus line may pass data in one direction only, or may be used to pass data in both directions. A unidirectional line is called a **simplex line**. A bidirectional line may carry data one direction at a time, in which case it is called a **half-duplex line**, or in both directions simultaneously, known as a **full-duplex line**. The same nomenclature is also used to describe data communication channels, because, ultimately, the basic concepts of bus lines and communication channels are essentially similar.

Buses are also characterized by the way that they interconnect the various components to which they are attached. A bus that carries signals from a single specific source to a single specific destination is identified as a **point-to-point bus**. Point-to-point buses that connect an external device to a connector are often referred to as **cables**, as in a printer cable or a network cable. Thus, the cable that connects the USB port in a personal computer from the computer to a printer is an example of a point-to-point bus. The internal connectors into which external cables can be plugged are often called **ports**. Typical ports on a personal computer might include parallel printer ports, network ports, USB ports, and firewire ports.

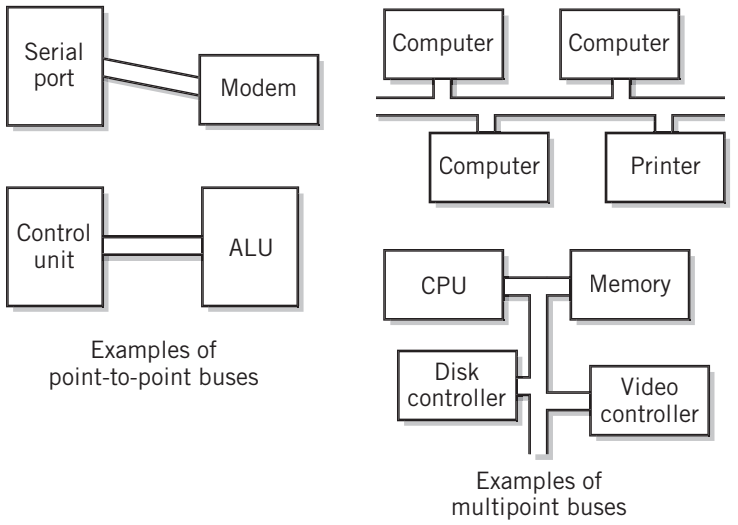
Alternatively, a bus may be used to connect several points together. Such a bus is known as a **multipoint bus**, or sometimes as a *multidrop bus*. It is also referred to as a **broadcast bus**, because the signals produced by a source on the bus are ''broadcast'' to every other point on the bus in the same way as a radio station broadcasts to anyone who tunes in. The bus in a traditional Ethernet network is an example of a broadcast bus: the signal being sent by a particular computer on the network is received by every other computer connected to the network. (The operation of Ethernet is discussed in Chapter 13.) In most cases, a multipoint bus requires addressing signals on the bus to identify the desired destination that is being addressed by the source at a particular time. Addressing is not required with a point-to-point bus, since the destination is already known, but an address may be required if the message is being passed *through* the destination point to another location. Addressing is also not required for a multipoint bus where the signal is actually intended to reach all the other locations at once; this is sometimes the case for buses that are internal to the CPU. Addressing may be integral to the lines of the bus itself, or may be part of the protocol that defines the meaning of the data signals being transported by the bus.

Typical point-to-point and multipoint bus configurations are illustrated in Figure 7.8

A parallel bus that carries, say, 64 bits of data and 32 bits of address on separate data and address lines would require a bus width of 96 lines, even before control lines are considered. The parallel bus is characterized by high throughput capability because all the bits of a data word are transferred at once. Virtually every bus internal to the CPU is a parallel bus, since the high speed is essential to CPU operation. Also most internal

FIGURE 7.8

Point-to-Point and Multipoint Buses



operations and registers are inherently parallel, and the use of serial buses would require additional circuitry to convert the parallel data to serial and back again. Until recently, the buses that connected the CPU with memory and various high speed I/O modules such as disk and display controllers were also parallel, for similar reasons.

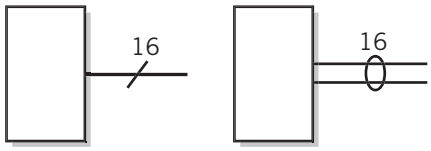
The parallel bus does have a number of disadvantages, though. Parallel buses are expensive and consume a considerable amount of space. Connectors used with parallel buses are also expensive because of the large number of pins involved. External parallel buses, such as printer cables are also expensive because of the large number of lines required. More seriously, parallel buses are subject to radio-generated electrical interference between the different lines at high data transfer rates. The higher the data rate, the worse the interference, which ultimately limits the speed at which the parallel bus can operate. Additionally, there is a slight difference in time delay on different lines, known as *skew*, as signals traverse the bus. The transfer rate, and thus the clock speed of the bus, is also limited by the requirement that the data must not change faster than the maximum skew time. Both of these problems can cause data corruption. Finally, the cost of fiber optic technology makes a parallel optical cable impractical.

Data on a serial bus is transferred sequentially, one bit at a time. Although you might think that the throughput of a serial bus would be lower than that of a parallel bus theoretically capable of the same per line transfer rate, the limitations noted above make serial bus transmission attractive in many circumstances. Indeed, with advances in serial bus technology, serial buses are now preferred for many, if not most, applications requiring high data transfer rates.

Generally, a serial bus has a single data line pair and perhaps a few control lines. (For simultaneous two-way communication, a second data line pair can be added.) There are

FIGURE 7.9

Alternative Bus Notations



no separate address lines in a serial bus. Serial buses are often set up for point-to-point connection; no addressing is required in this case. If addressing is required in a serial bus application, the address may be **multiplexed** with the data. What this means is that the same line is used for both address and data at different times; if an address is required, for example, the address might be sent first, one bit at a time, followed by the data. At its simplest, the serial bus can be reduced to a single data line pair, used for data, control, and addressing. Using modern materials such as fiber optics, very high transfer rates may be achieved. In general, control is handled using a bus protocol that establishes agreement as to the meaning and timing of each signal on the line among the components connected to the line.

It is also possible to design a parallel bus that multiplexes addresses and data on the same lines, as the PCI bus does, or multiplexes 32-bit data on sixteen data lines, for example. For example, the Pentium 4 multiplexes 128-bit data words to fit a 64-bit data path on the Pentium system bus.

To use a bus, the circuits that are connected to the bus must agree on a **bus protocol**. Recall from Chapter 1 that a protocol is an agreement between two or more entities that establishes a clear, common path of communication and understanding between them. A bus protocol is simply a specification that spells out the meaning of each line and each signal on each line for this purpose. Thus, a particular control line on a bus might be defined as a line that determines if the bus is to be used for memory read or memory write. Both the CPU and memory would have to agree, for example, that a "0" on that particular line means "memory read" and a "1" on the line means "memory write". The line might have a name like MREAD/$\overline{\text{MWRITE}}$, where the bar over MWRITE means that a "0" is the active state. The bar itself stands for "NOT".[3]

Buses are frequently notated on diagrams using widened lines to indicate buses. Sometimes a number is also present on the diagram. The number indicates the number of separate lines in the bus. Two alternative ways of notating buses in diagrams are shown in Figure 7.9.

7.6 CLASSIFICATION OF INSTRUCTIONS

Nearly every instruction in a computer performs some sort of operation on one or more source data values, which results in one or more destination data values. The operation may be a move or load, it may be an addition or subtraction, it may be an input or output, or it may be one of many other operations that we have already discussed.

[3] A pound sign (#) following the name is sometimes used to stand for "NOT" instead.

Actually, if you think about the classes of instructions that we have discussed, you will realize that there are only a very few instructions that do *not* operate on data. Some of these are concerned with the flow of the program itself, such as unconditional JUMP instructions. There are also instructions that control the administration of the computer itself; the only example in the Little Man Computer instruction set is the COFFEE BREAK or HALT that causes the computer to cease executing instructions. Another example on many computers is the NO OPERATION instruction that does nothing but waste time (which can be useful when a programmer wants to create a time delay for some reason).

Most modern computers also provide instructions that aid the operating system software in its work, by providing security, controlling memory access, and performing other functions. Because the operating system will frequently be controlling many tasks and users, these instructions must not be available to the users' application programs. Only the operating system can execute these instructions. These instructions are known as **privileged instructions**. The HALT instruction is usually a privileged instruction, because you would not want an individual user to stop the computer while other users are still in the middle of their tasks.

Computer manufacturers usually group the instruction set into various categories of instructions, such as data movement instructions, arithmetic instructions, shift and rotate instructions, input/output instructions, conditional branch instructions, jump instructions, and special-purpose instructions.

Within each category, the instructions usually have a similar instruction word format, support similar addressing modes, and execute in a similar way. A typical instruction set, divided into eight categories, appears in Figure 7.10. This figure represents nearly all the user-accessible instructions in the Motorola 68000 series of microprocessors used in early Apple Macintosh computers.[4] The privileged instructions are not listed in the diagram, nor are exception-handling instructions that are used primarily by system programmers. These constitute an additional two categories for the 68000 series CPUs. Incidentally, notice that this CPU does not have any I/O instructions. That is because the CPU is designed in such a way that the move instructions can also be used for I/O. Notice particularly that, except for the lack of I/O instructions, the categories conform fairly well to the Little Man Computer instruction set. The additional instructions in this CPU are mostly variations on instructions that are familiar to you plus special control instructions. The 68000 series CPUs also support a math coprocessor, which adds a category of floating point arithmetic instructions. The floating point math instructions are built directly into 68000 series processors starting with the 68040 CPU.

Data Movement Instructions (LOAD, STORE, and Other Moves)

Because the move instructions are the most frequently used, and therefore the most basic to the computer, computer designers try to provide a lot of flexibility in these instructions. The MOVE category commonly includes instructions to move data from memory to general registers, from general registers to memory, between different general registers, and, in

[4] Although the 68000 CPU series is old, it is still used in embedded computer systems. It was selected for this illustration because of its clean design, with few extraneous bells and whistles.

FIGURE 7.10

68000 Instruction Set

Mnemonic	Operation
	Data Movement Instructions
CAS*	Compare and swap with operand
CAS2*	Compare upper/lower and swapASR
EXG	Exchange registers
LEA	Load effective address
LINK	Link and allocate stack
MOVE	Move src to dst
MOVE16	Move src to dst (68030-68060 only)
MOVEA	Move src to address register
MOVEM	Move multiple registers at once
MOVEP	Move to peripheral
MOVEQ	Move short data to dst
PEA	Push effective address to stack
UNLK	Unlink stack
	Integer Arithmetic Instructions
ADD	Add src to dst
ADDA	Add src to address register
ADDI	Add immediate data to dst
ADDQ	Add short data to dst
ADDX	Add with extend bit to dst
SUB, SUBA, SUBI, SUBQ, SUBX	Subtracts act similarly to adds
MULS	Signed multiply
MULU	Unsigned multiply
DIVS	Signed divide
DIVU	Unsigned divide
DIVSL*	Long signed divide
DIVUL*	Unsigned long divide
CLR	Clear value in register
CMP	Compare src to dst
CMPA	Compare src to address register
CMPI	Compare immediate data to dst
CMPM	Compare memory
CMP2*	Compare register to upper/lower bounds
EXT	Sign extend
EXTB	Sign extend byte
NEG	Negate register
NEGX	Negate with extend
	Boolean Logic Instructions
AND	AND src to dst
ANDI	AND immediate data to dst
EOR	Exclusive OR src to dst
EORI	Exclusive OR immediate data to dst
NOT	NOT destination
OR	OR src to dst
ORI	OR immediate data to dst
Scc	Test condition codes and set operand
TAS	Test and set operand
TST	Test operand and set condition codes
TRAPcc*	Trap on condition

Mnemonic	Operation
	Shift and Rotate Instructions
ASL	Arithmetic shift register left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right
ROXL	Rotate left with extend bit
ROXR	Rotate right with extend bit
SWAP	Swap words of a long word
	Bit Manipulation Instructions
BCHG	Change bit
BCLR	Clear bit
BTEST	Set bit
BTST	Test bit
	Bit Field Instructions
BFCHG*	Change bit field
BFCLR*	Clear bit field
BFEXTS*	Extract and sign extend bit field
BFEXTU*	Extract and zero extend bit field
BFFFO*	Find first set bit in bit field
BFINS*	Insert bit field
BFSET*	Set bit field
BFTST*	Test bit field
	Binary Coded Decimal Instructions
ABCD	Add src to dst
NBCD	Negate destination
PACK*	Pack src to dst
SBCD*	Subtract src from dst
UNPK*	Unpack src to dst
	Program Flow Instructions
Bcc	Branch on condition code cc
BRA	Branch unconditionally
BSR	Branch to subroutine
CALLM*	Call module
DBcc	Test, decrement, and branch on condition
JMP	Jump to address
JSR	Jump to subroutine
NOP	No operation
RTD*	Return and deallocate stack (also 68010)
RTE	Return from exception (privileged)
RTM*	Return from module
RTR	Return and restore condition codes
RTS	Return from subroutine
TRAP	Trap to system

*(68020–68060 only)

(src = source; dst = destination; cc = condition code indicator, e.g. BGT branch of greater than)

some computers, directly between different memory locations without affecting any general register. There may be many different addressing modes available within a single computer.

Additionally, variations on these instructions are frequently used to handle different data sizes. Thus, there may be a LOAD BYTE instruction, a LOAD HALF-WORD (2 bytes), a LOAD WORD (4 bytes), and a LOAD DOUBLE WORD (8 bytes) within the same instruction set. (Incidentally, the concept of a "word" is not consistent between manufacturers. To some manufacturers the size of a word is 16 bits; to others, it is 32 or even 64 bits).

The Little Man LOAD and STORE instructions are simple, though adequate, examples of MOVE instructions. Other than expanding the addressing mode capabilities and adding multiple word size capabilities, which we have already discussed, the major limitation of the Little Man LOAD and STORE instructions is the fact that they are designed to operate with a single accumulator.

When we expand the number of accumulators or general-purpose registers, we must expand the instruction to determine which register we wish to use. Thus, the instruction must provide a field for the particular register. Fortunately, it takes very few bits to describe a register. Even sixteen registers require only 4 bits. On the other hand, if the computer uses the registers to hold pointers to the actual memory addresses as its standard addressing mode, the required instruction size may actually decrease, since fewer bits are required for the address field in this case.

Additionally, it is desirable to have the capability to move data directly between registers, since such moves do not require memory access and are therefore faster to execute. In fact, some modern CPUs, including the Sun SPARC and IBM PowerPC architectures, provide only one pair of LOAD/STORE or MOVE instructions for moving data between the CPU and memory. All other instructions in these CPUs move and manipulate data only between registers. This allows the instruction set to be executed much more rapidly. There is a detailed examination of the Power PC computer and its variants in Supplementary Chapter 2.

Arithmetic Instructions

Every CPU instruction set includes integer addition and subtraction. Except for a few special-purpose CPUs, every CPU today also provides instructions for integer multiplication and division. Many instruction sets provide integer arithmetic for several different word sizes. As with the MOVE instructions, there may be several different integer arithmetic instruction formats providing various combinations of register and memory access in different addressing modes.

In addition, most current CPUs also provide floating point arithmetic capabilities. On older PCs with 80386 or earlier processors, a floating point math coprocessor unit had to be purchased separately and installed in a socket provided for that purpose on the motherboard of the computer. Because of the expense, most users would not exercise this option. Extensive floating point calculations are required for many graphics applications, such as CAD/CAM programs, animation, and computer games; the presence of floating point instructions reduces the processing time significantly. Floating point instructions usually operate on a separate set of floating point data registers with 64-, 80-, or 128-bit word sizes. The modern instruction set usually also contains instructions that convert data between integer and floating point formats.

As noted in Chapter 5, most modern CPUs also provide at least a minimal set of arithmetic instructions for BCD or packed decimal format, which simplifies the programming of business data processing applications.

Of course, it is not absolutely necessary to provide all these different instruction options. Multiplication and division can be performed with repeated addition and subtraction, respectively. In computers there is an even easier technique. In elementary school you probably learned the "long" multiplication and division methods which multiply or divide numbers one digit at a time and shift the results until the entire operation is complete. Because of the simplicity of binary multiplication ($1 \times 1 = 1$, all other results are 0), the computer can implement the same method using only adds or subtracts together with shift instructions. Internally, the multiplication and division instructions simply implement in hardware this same method. Since the fetch-execute cycle requires a single-bit shift and register add step for each bit in the multiplier, multiply and divide instructions execute slowly compared to other instructions.

Even the subtract instruction is theoretically not necessary, since we showed in Chapter 4 that integer subtraction is performed internally by the process of complementing and adding.

As we already noted, the same is true of BCD and floating point instructions. On the now rare computers that do not provide floating point instructions, there is usually a library of software procedures that are used to simulate floating point instructions.

Boolean Logic Instructions

Most modern instruction sets provide instructions for performing Boolean algebra. Commonly included are a NOT instruction, which inverts the bits on a single operand, as well as AND, (inclusive) OR, and EXCLUSIVE-OR instructions, which require two source arguments and a destination.

Single Operand Manipulation Instructions

In addition to the NOT instruction described in the previous paragraph, most computers provide other convenient single operand instructions. Most of these instructions operate on the value in a register, but some instruction sets provide similar operations on memory values as well. Most commonly, the instruction set will contain instructions for NEGATing a value, for INCREMENTing a value, for DECREMENTing a value, and for setting a register to zero. There are sometimes others. On some computers, the increment or decrement instruction causes a branch to occur automatically when zero is reached; this simplifies the design of loops by allowing the programmer to combine the test and branch into a single instruction.

Bit Manipulation Instructions

Most instruction sets provide instructions for setting and resetting individual bits in a data word. Some instruction sets also provide instructions for operating on multiple bits at once. Bits can also be tested, and used to control program flow. These instructions allow programmers to design their own "flags" in addition to commonly provided negative/positive, zero/nonzero, carry/borrow, and overflow arithmetic flags.

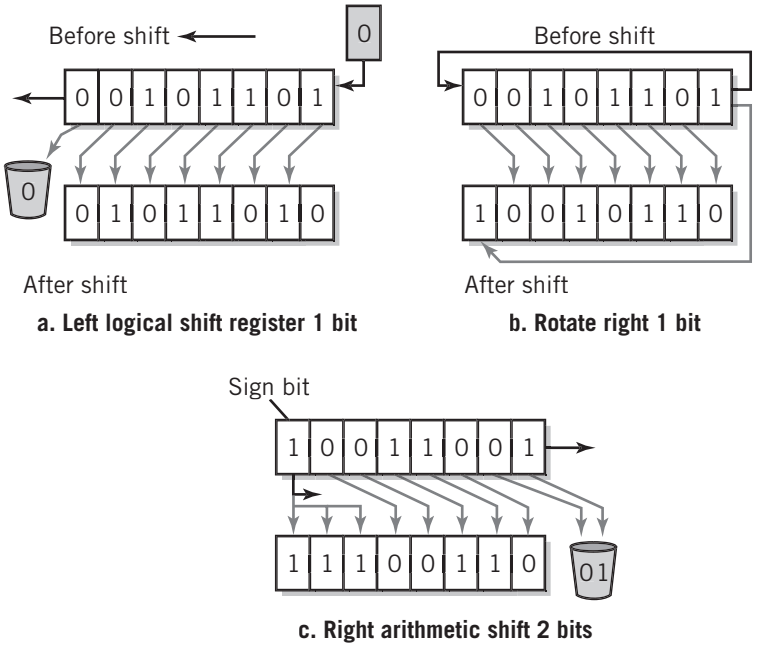
Shift and Rotate Instructions

Shift and **rotate operations** have been mentioned previously as a means to implement multiplication and division. Shifts and rotate operations have other programming applications, and CPU instruction sets commonly provide a variety of different shift and rotate instructions for the programmer to use. As shown in Figure 7.11, shift instructions move the data bits left or right one or more bits. Rotate instructions also shift the data bits left or right, but the bit that is shifted out of the end is placed into the vacated space at the other end. Depending on the design of the particular instruction set, bits shifted out the end of the word may be shifted into a different register or into the carry or overflow flag bit, or they may simply "fall off the end" and be lost.

Two different kinds of shifts are usually provided. The data word being shifted might be logical or it might be numeric. **Logical shift** instructions simply shift the data as you would expect, and zeros are shifted in to replace the bit spaces that have been vacated. **Arithmetic shift** instructions are commonly used to multiply or divide the original value by a power of 2. Therefore, the instruction does not shift the leftmost bit, since that bit usually represents the algebraic sign of the numeric value—obviously the sign of a number must be maintained. Left arithmetic shifts do not shift the left bit, but zeros replace the bits from the right as bits are moved to the left. This will effectively double the numeric value for each shift of one bit. On the other hand, right arithmetic shifts fill the space of moved bits with the sign bit rather than with zero. This has the effect of halving the value

FIGURE 7.11

Typical Register Shifts and Rotates



a. Left logical shift register 1 bit

b. Rotate right 1 bit

c. Right arithmetic shift 2 bits

for each bit shifted, while maintaining the sign of the value. It may not seem obvious to you that this works correctly, but it becomes more apparent if you recall that negative numbers in complementary arithmetic count backward starting from the value −1, which is represented in 2's complement by all ones.

Rotate instructions take the bits as they exit and rotate them back into the other end of the register. Some instructions sets include the carry or overflow bit as part of the rotation. Some CPUs also allow the rotation to take place between two registers. Rotate instructions can be used to exchange the 2 bytes of data in a 16-bit word, for example, by rotating the word by 8 bits.

Program Control Instructions

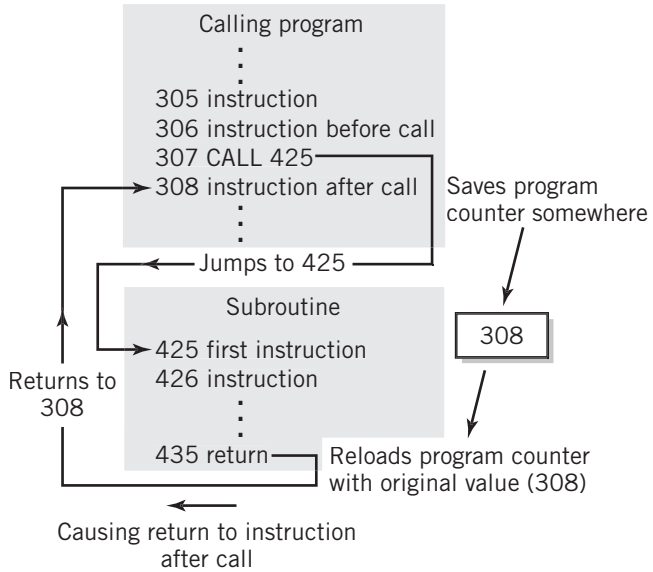
Program control instructions control the flow of a program. Program control instructions include jumps and branches, both unconditional and conditional, and also **subroutine CALL and RETURN** instructions. Various conditional tests are provided, including those with which you are already familiar: branch on zero, branch on nonzero, branch on positive, branch on negative, branch on carry, and so on.

CALL instructions, sometimes known as JUMP SUBROUTINE instructions, are used to implement subroutine, procedure and function calls. Thus, CALL instructions are important as a means to enable program modularization.

From your programming experience, recall what happens when your program calls a subroutine or procedure. The program jumps to the starting location of the subroutine and executes the code in the subroutine. When the subroutine is completed, program execution returns to the calling program and continues with the instruction following the call. The machine language CALL instruction works the same way. A jump to the starting location of the subroutine occurs, and execution continues from that point. The only difference between a CALL instruction and a normal JUMP instruction is that the CALL instruction must also save somewhere the program counter address from which the jump occurred, so that the program may return to the instruction in the calling program following the call after the subroutine is completed. The RETURN instruction restores the original value to the program counter, and the calling program proceeds from where it left off. Operation of the CALL and RETURN instructions are illustrated in Figure 7.12.

FIGURE 7.12

Operation of CALL and RETURN Instructions



Different computers use different methods to save the return address. One common method is to store the return address on a memory stack; the RETURN instruction operates by removing the address from the stack and moving it to the program counter. The use of stacks is discussed briefly in the next section. Another method for performing CALLS and RETURNS is explored in Exercise S3.14.

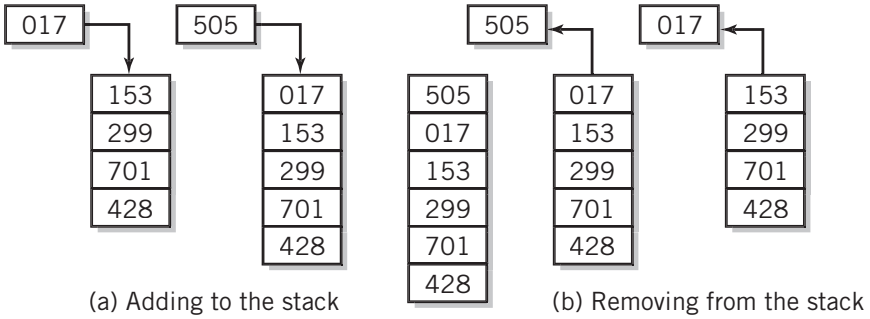
Stack Instructions

One of the most important data storage structures in programming is the **stack**. A stack is used to store data when the most recently used data will also be the first needed. For that reason, stacks are also known as *LIFO*, for *l*ast-*i*n, *f*irst-*o*ut, structures. As an analogy, stacks are frequently described by the way plates are stored and used in a cafeteria. New plates are added to the top of the stack, or *pushed*, and plates already on the stack move down to make room for them. Plates are removed from the top of the stack, or *popped*, so that the last plates placed on the stack are the first removed. Similarly, the last number entered onto a computer memory stack will be the first number available when the stack is next accessed. Any data that must be retrieved in reverse order from the way it was entered is a candidate for the use of stacks. Figure 7.13 shows the process of adding to and removing numbers from the stack.

Stacks are an efficient way of storing intermediate data values during complex calculations. In fact, storage in Hewlett-Packard calculators is organized around a stack of memory. As we already noted, stacks are also an excellent method for storing the return addresses and arguments from subroutine calls. Program routines that are recursive must "call themselves." Suppose the return address were stored in a fixed location, as shown in Figure 7.14a. If the routine is called a second time, from within itself, Figure 7.14b, the original returning address (56) is lost and replaced by the new return address (76). The program is stuck in an infinite loop between 76 and 85. In Figure 7.15, the return address is stored on a stack. This time when the routine is again called, the original address is simply

FIGURE 7.13

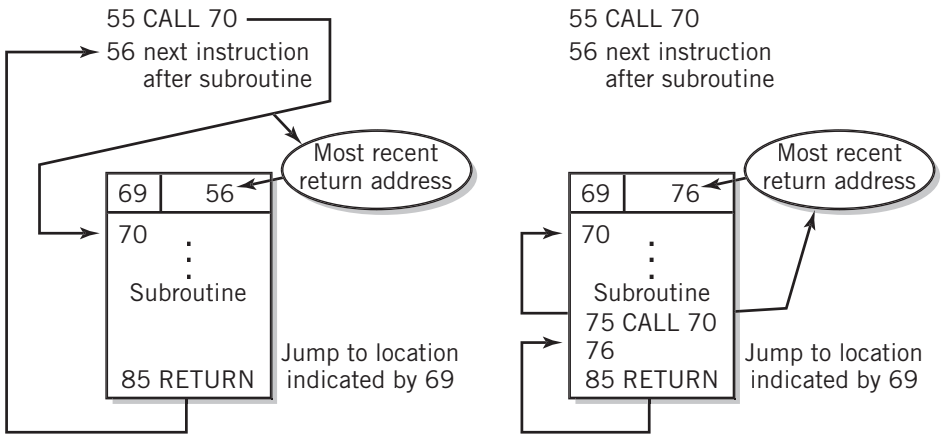
Using a Stack



(a) Adding to the stack (b) Removing from the stack

FIGURE 7.14

Fixed Location Subroutine Return Address Storage



a. Subroutine called from loc.55

b. Subroutine re-called from 75, within the subroutine

pushed down the stack, below the most recent address. Notice that the program "winds its way back out" in the reverse order from which the routines were entered. This is exactly what we want: we always return from the last called subroutine to the one just previous. J. Linderman of Bentley University notes that the same technique would be used to back out of a maze for which the explorer has written down each turn that she made after entering.

There are many other interesting applications for stacks in computers, but further discussion is beyond the scope of this book. The curious reader is referred to *For Further Reading* for references.

Computers do not generally provide special memory for stack use, although many machines provide special STACK instructions to simplify the bookkeeping task. Instead, the programmer sets aside one or more blocks of regular memory for this purpose. The "bottom" of the stack is a fixed memory location, and a **stack pointer** points to the "top" of the stack, that is, the most recent entry. This is shown in Figure 7.16. A new entry is added to the stack, or pushed, by incrementing the stack pointer, and then storing the data at that location. An entry is removed from the stack, or popped, by copying the value pointed to and then decrementing the stack pointer. If a register is provided for the stack pointer, register-deferred addressing can be used for this purpose. (You should note that memory is drawn upside-down in Figure 7.16 so that incrementing the stack pointer moves it *upward*.)

Many instruction sets provide PUSH and POP instructions as direct support for stacks, but stacks can be implemented easily without special instructions. (Exercise S3.15 illustrates one solution.) Some computers also specify the use of a particular general-purpose register as a stack pointer register.

FIGURE 7.15

Stack Subroutine Return Address Storage

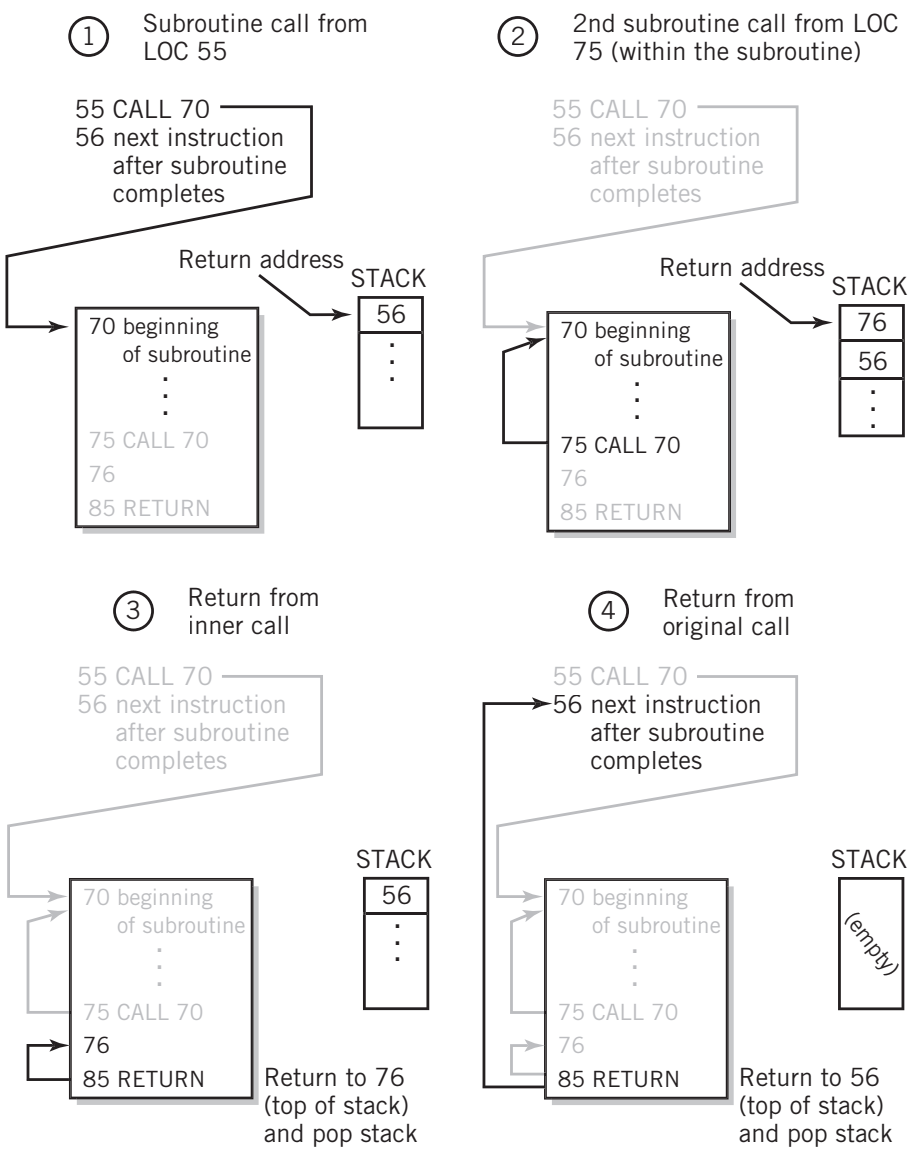
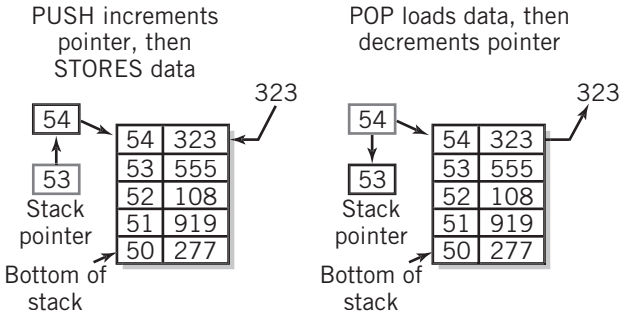


FIGURE 7.16

Using a Block of Memory as a Stack



Multiple Data Instructions

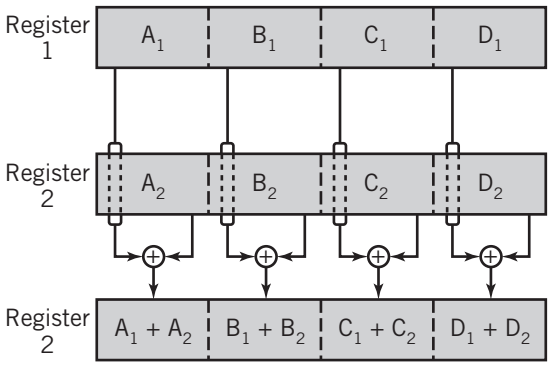
Multimedia applications rank high in computational demands on the CPU in modern PCs and workstations. In response to the demand, CPU designers have created specialized instructions that speed up and simplify multimedia processing operations.

Multimedia operations are commonly characterized by a number of simple operations applied identically to every piece of data in the set. As a simple example, the brightness of an image might be modified by multiplying the value of every pixel in the image by a common scale factor. Or, a measure of similarity between two images could be established by subtracting all the pixel values in one image from the corresponding pixel values in a second image and averaging the results.

Multiple data instructions perform a single operation on multiple pieces of data simultaneously. For this reason they are also known as **SIMD** instructions. SIMD stands for **S**ingle **I**nstruction, **M**ultiple **D**ata. The SIMD instructions provided on Intel Pentium processors are typical. The processor provides eight 128-bit registers specifically for SIMD instruction use and also allows the use of the standard 64-bit floating point registers for this purpose. The Pentium CPU SIMD instructions can process from two 64-bit integers up to sixteen 8-bit integer arithmetic operations or up to two 64-bit floating point number operations simultaneously as well as providing instructions for packing and unpacking the values and moving them between registers and memory, and a variety of other related instructions. Other vendors, including AMD, IBM, Sun, Transmeta, and VIA provide compatible or similar SIMD instructions. The IBM Cell processor, which serves as the CPU in the Sony Playstation 3, provides a particularly powerful SIMD capability that accounts for much of the Playstation's graphics strength. Figure 7.17 shows the operation of a SIMD ADD instruction.

FIGURE 7.17

Operation of a 4-Wide SIMD ADD Instruction



Although multimedia operations are a primary application for these instructions, these instructions can be applied to any vector or array processing application, and are useful for a number of purposes in addition to multimedia processing, including voice-to-text processing, the solutions to large-scale economics problems, weather prediction, and data encryption and decryption.

Other Instructions

The remainder of the instructions includes input/output instructions and machine control instructions. In most systems both groups are privileged instructions. Input/output instructions are generally privileged instructions because we do not want input and output requests from different users and programs interfering with each other. Consider, for example, two users requesting printer output on a shared printer at the same time, so that each page of output is divided back and forth between the two users. Obviously, such output would not be acceptable. Instead, these requests would be made to the operating system that controls the printer, which would set priorities, maintain queues, and service the requests. We will deal with the subject of input/output in Chapters 9 and 10, and with operating systems in Chapters 15 through 18.

7.7 INSTRUCTION WORD FORMATS

Instructions in the Little Man Computer were made up entirely of three-digit decimal numbers, with a single-digit op code, and a two-digit **address field**. The address field was used in various ways: for most instructions, the address field contained the two-digit address where data for the instruction could be found (e.g., LOAD) or was to be placed (STORE). In a few instructions, the address field was unused (e.g., HALT). For the branch instructions, the address field space was used instead to hold the address of the next instruction to be executed. For the I/O instructions, the address field became a sort of extension of the op code. In reality, the I/O address field contained the "address" of an I/O device, in our case 01 for the in basket and 02 for the out basket.

The instruction set in a typical real CPU is similar. Again, the instruction word can be divided into an op code and zero or more address fields. A simple 32-bit instruction format with one address field might look like that shown in Figure 7.18. In this example, the 32 bits are divided into an 8-bit op code and 24 bits of address field.

In the Little Man Computer, reference to an *address* specifically referred to a *memory* address. However, we have already noted that the computer might have several general-purpose registers and that it would be necessary for the programmer to select a particular register to use as a part of the instruction. To be more general, we will use the word "address" to refer to any data location, whether it is a user-accessible register or a memory location. We will use the more specific expression *memory address* when we want to specify that the address is actually a memory location.

In general, computer instructions that manipulate data require the specification of at least two locations for the data: one or more *source* locations and one *destination* location. These locations may be expressed *explicitly*, as address fields in the instruction word, or *implicitly*, as part of the definition of the instruction itself. The instruction format of the Little Man LOAD instruction, for example, takes the data from the single address field as the

FIGURE 7.18

A Simple 32-bit Instruction Format

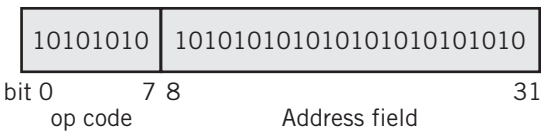


FIGURE 7.19

Typical Two Operation Register Move Format

op code	source register	destination register
MOVE	5	10

explicit source address. Explicit addresses in the Little Man Computer are always memory addresses. The destination address in this case is **implicit**: this instruction always uses the accumulator register as a destination. The Little Man ADD and SUBTRACT instructions require *two* sources and a destination. Source data addressed by the instruction's single explicit address field is added to the value in the implicitly stated accumulator, with the result placed implicitly in the accumulator.

For a particular instruction, the source(s) and destination may be the same or may be different. For example, an instruction that complements a value to change its sign would usually be done "in place"; that is, the source and destination register or memory location is usually the same. The Little Man ADD instruction uses the accumulator register both as a source for one of the numbers to be added and as the destination for the result. On the other hand, when we move data, using a LOAD or STORE or some other type of MOVE operation, two operands are required. The source and destination are obviously different, or the move would not be useful! A register-to-register MOVE, for example, might use an instruction format such as that shown in Figure 7.19. In the figure, the instruction word consists of an opcode and two fields that point to registers. As shown, this instruction would move data from register 5 to register 10. Unless the operation is done in place, the sources are normally left unchanged by the instruction, whereas the destination is almost always changed.

The source and destination addresses may be registers or memory locations. Since most modern computers have multiple registers available to the user it is usually necessary to provide at least two explicit address fields, even for an address-register move, since the number of the particular register must be specified in the instruction.

The sources and destinations of data for an instruction, whether implicit or explicit, are also known as **operands**. Thus, instructions that move data from one place to another have two operands: one source operand and one destination operand. Arithmetic operations such as ADD and SUBTRACT require three operands. Explicit address fields are also known as *operand fields*.

Most commonly, instructions that manipulate data will have one address field for operations that happen in place, and two or three address fields for move and arithmetic operations. On some computers one or more of the addresses may be implicit, and no address field is required for the implicit address. However, in modern computers most address references are explicit, even for register addresses, because this increases the generality and flexibility of the instruction. Thus, most computer instructions will consist of an op code and one, two, or three explicit address fields. Some textbooks refer to instructions with one, two, or three explicit address fields as *unary*, *binary*, or *ternary* instructions, respectively.

7.8 INSTRUCTION WORD REQUIREMENTS AND CONSTRAINTS

The size of the instruction word, in bits, is dependent on the particular CPU architecture, particularly by the design of its instruction set. The size of the instruction word may be

fixed at, say, 32 bits, or it may vary depending on the usage of the address fields. The Sun Sparc CPU, for example, takes the former approach: every instruction word is exactly 32 bits wide. Conversely, some of the basic instruction words for the x86 microprocessor line used in the common PC, for example, are as small as 1 or 2 bytes long, but there are some instructions in the Pentium microprocessor that are as many as 15 bytes long. The IBM Series z architecture is an evolutionary extension of upward compatible CPU architectures dating back to the 1960s. The legacy instructions in the IBM Series z CPU are mostly 4 bytes, or 32 bits long, with a few 2-byte or 6-byte long instructions. To expand the architecture to 64-bit addressing and data, IBM added a number of new instructions. These are all 6 bytes in length.

The challenge in establishing an instruction word size is the need to provide both enough op code bits to support a reasonable set of different instructions as well as enough address field bits to meet the ever growing demand for increasing amounts of addressable memory. Consider again, for example, the extremely straightforward instruction format shown in Figure 7.18. This format assumes a single address field with a 32-bit fixed length instruction. With the division shown, we have access to $2^8 = 256$ different instructions and $2^{24} =$ approximately 16 million memory addresses.

Even if the designer creates a smaller instruction set, with fewer op codes, the amount of memory that may be specified in a 32-bit instruction word is severely limited by modern standards. Most of today's computers support an address size of at least 32 bits. Many newer machines support 64-bit addresses.

Further, with additional registers, the simple instruction format shown in Figure 7.18 must be expanded to handle explicit addressing of multiple registers, including moves between registers, as well as identifying the proper register in operations between registers and memory. In short, the simple instruction format used in the Little Man Computer is inadequate for the instruction sets in modern computers.

The use of instructions of different lengths is one of several techniques developed by instruction set designers to allow more flexibility in the design of the instruction set. Simple instructions can be expressed in a small word, perhaps even a single byte, whereas more complicated instructions will require instruction words many bytes long. Longer instructions are stored in successive bytes of memory. Thus, a Little Man HALT, IN, or OUT instruction would be stored in a single location. A LOAD might require two successive locations to store memory addresses of five digits or three locations for an eight-digit address. The use of variable length instructions is efficient in memory usage, since each instruction is only as long as it needs to be.

There are a number of important disadvantages to variable length instructions, however. Most modern computers increase CPU processing speed by ''pipelining'' instructions, that is, by fetching a new instruction while the previous one is still completing execution, similar to the processing on an automobile assembly line. Variable length instructions complicate pipelining, because the starting point of the new instruction is not known until the length of the previous instruction has been determined. If you extend this idea to multiple instructions, you can see the difficulty of maintaining a smooth assembly line. This issue is discussed in more detail in Chapter 8. Because pipelining has become so important to processing speed in modern computers, the use of variable length instructions has fallen out of favor for new CPU designs. Nearly all new CPU designs use fixed length instructions exclusively.

As we mentioned previously in our discussion of memory size, an effective alternative to large instructions or variable instruction words is to store the address that would otherwise be located in an instruction word address field at some special location that can hold a large address, such as a general purpose register, and use a small address field within the instruction to point to the register location. There are a number of variations on this theme. This technique is used, even on systems that provide variable length instructions. A single CPU might provide a number of different variations to increase the flexibility of the instruction set. This flexibility also includes the ability to code programs that process lists of data more efficiently. The various ways of addressing registers and memory are known as addressing modes. The Little Man Computer provides only a single mode, known as direct addressing. The alternative just described is called register deferred addressing. An example of a deferred LOAD instruction is shown in Figure 7.20. This instruction would load the data value stored at memory address 3BD421 into general purpose register 7. There are a number of addressing modes discussed in detail in Supplementary Chapter 3. The use of different addressing modes is the most important method for minimizing the size of instruction words and for writing efficient programs.

Examples of instruction formats from two different CPUs are shown in Figure 7.21. There may be several different formats within a single CPU. We have shown only a partial set for each machine, although the SPARC set is complete except for small variations. (There are twenty-three different IBM formats in all.) It is not necessary that you understand every detail in Figure 7.21, but it is useful to note the basic similarities between the instruction set formats in different computers.

FIGURE 7.20

Deferred Register Addressing

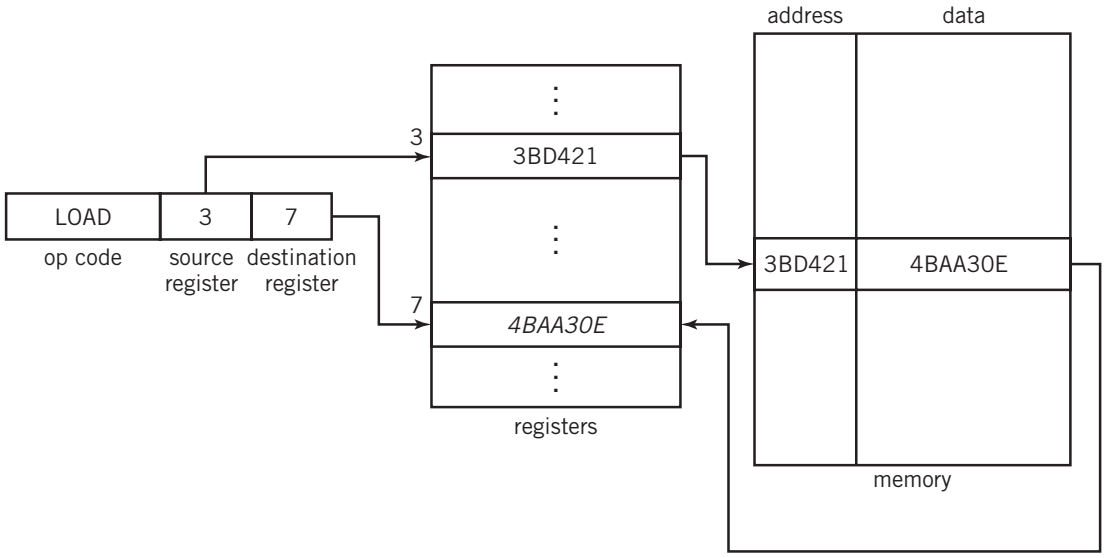
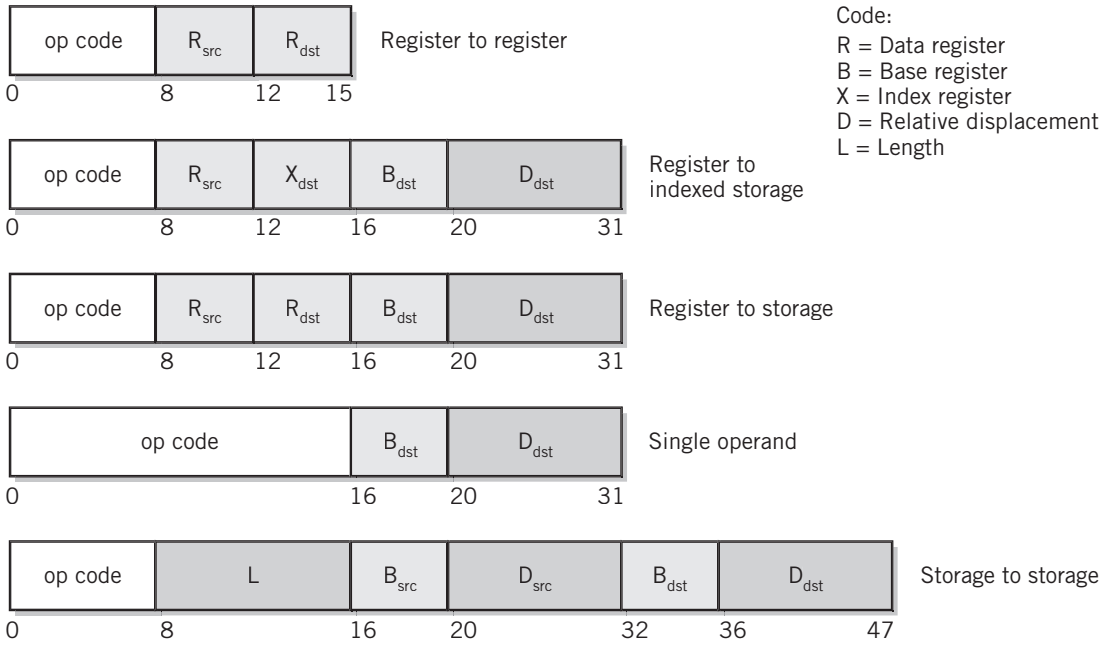
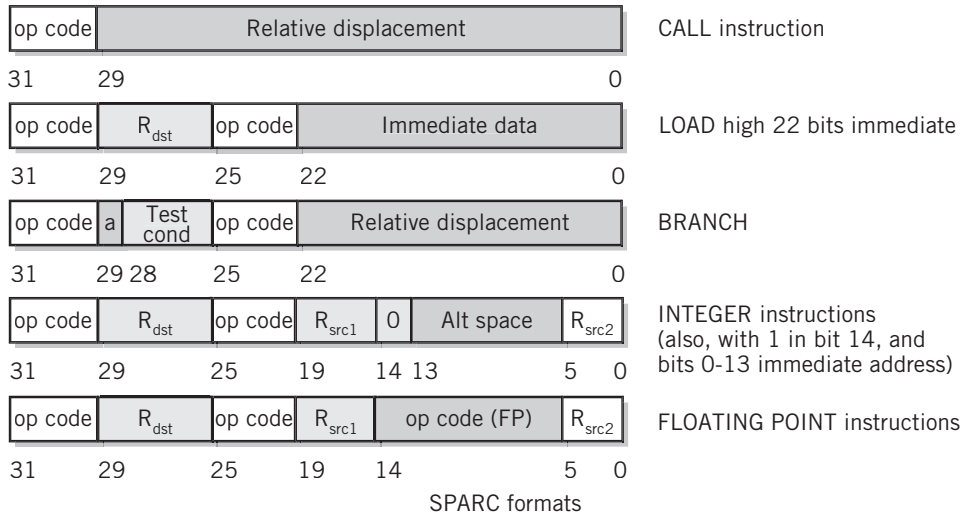


FIGURE 7.21

Examples of Instruction Formats



IBM mainframe formats



SPARC formats

SUMMARY AND REVIEW

Functionally, the operation of the CPU, together with memory, is essentially identical to that of the Little Man Computer. For each component of the Little Man Computer, there is a corresponding component in the computer unit.

Within the CPU, the most important components are registers. Data may be moved between registers, may be added or subtracted from the current contents of a register, and can be shifted or rotated within a register or between registers. Each instruction in the instruction set is executed by performing these simple operations, using the appropriate choice of registers and operations in the correct sequence for the particular instruction. The sequence of operations for a particular instruction is known as its fetch-execute cycle. A fetch-execute cycle exists for every instruction in the instruction set. Fetch-execute instruction cycles constitute the basis for all program execution in the computer. The sequence for each instruction corresponds closely to the actions taken by the Little Man in performing a similar instruction.

The operation of memory is intimately related to two registers in particular, the memory address register and the memory data register. Addresses placed into the MAR are decoded in memory, resulting in the activation of a single memory address line. At the proper instant, data can then be transferred in either direction between that memory location and the MDR. The direction is specified by a read/write control line. The number of available memory locations is established by the size of the MAR; the data word size is established by the size of the MDR.

Interconnections between various parts of a computer are provided by buses. There are many different types of buses. Buses connect different modules within the CPU. They also connect the CPU to memory and to the I/O peripherals. Buses can connect two components in a point-to-point configuration or may interconnect several modules in a multipoint configuration. Buses may be parallel or serial. In general, the lines on buses carry signals that represent data, address, and control functions.

Instructions fall naturally into a small number of categories: moves, integer arithmetic, floating point arithmetic, data flow control, and so forth. There are also privileged instructions, which control functions internal to the CPU and are accessible only to the operating system.

Instructions in a real CPU are made up of an op code and up to three address field operands. The size of the instruction word is CPU dependent. Some computers use variable length instruction words. Other computers use a fixed length instruction, most commonly, 32 bits in length.

FOR FURTHER READING

There are many excellent textbooks that describe the implementation and operation of the components of the computer system. A brief, but very clear, explanation of the fetch-execute cycle can be found in Davis and Rajkumar [DAV02]. Three classic engineering textbooks that discuss the topics of this chapter in great detail are those authored by Stallings [STAL05], Patterson and Hennessy [PATT07], and Tanenbaum [TAN05]. Wikipedia offers a brief, but clear, introduction to the principal concepts of von Neumann architecture. There are many books and papers describing various components and techniques associated

with the implementation and operation of the CPU and memory. Also see the For Further Reading section in Chapter 8 for more suggestions.

KEY CONCEPTS AND TERMS

accumulator
address field
arithmetic/logic unit (ALU)
arithmetic shift
broadcast bus
bus
bus interface bridge
bus protocol
cable
central processing unit (CPU)
control unit (CU)
dynamic RAM
EEPROM (electronically erasable programmable ROM)
explicit source address
fetch-execute instruction cycle
flag
flash memory
full-duplex line
general-purpose register
half-duplex line
implicit source address
instruction pointer
instruction register (IR)
line (bus)
logical shift
memory
memory address register (MAR)
memory data register (MDR)
memory management unit
multiplex
multipoint bus
nonvolatile memory
operands
parallel bus
point-to-point bus
port
program counter (PC)
program counter register
Program Status Word (PSW)
privileged instruction
RAM
register
register file
ROM
rotate operation
serial bus
shift operation
SIMD
simplex line
stack
stack pointer
static RAM
status register
subroutine call and return
user-visible register
volatile memory

READING REVIEW QUESTIONS

7.1 What does *ALU* stand for? What is its corresponding component in the Little Man Computer? What does *CU* stand for? What is its corresponding LMC component?

7.2 What is a *register*? Be precise. Name at least two components in the LMC that meet the qualifications for a register. Name several different kinds of values that a register might hold.

7.3 What is the purpose of the *instruction register*? What takes the place of the instruction register in the LMC?

7.4 When a value is copied from one register to another, what happens to the value in the source register? What happens to the value in the destination register?

7.5 There are four primary operations that are normally performed on a register. Describe each operation.

7.6 Explain the relationship between the memory address register, the memory data register, and memory itself.

7.7 If the memory register for a particular computer is 32 bits wide, how much memory can this computer support?

7.8 What is the difference between volatile and nonvolatile memory? Is RAM volatile or nonvolatile? Is ROM volatile or nonvolatile?

7.9 Registers perform a very important role in the fetch-execute cycle. What is the function of registers in the fetch-execute instruction cycle?

7.10 Explain each of the fetch part of the fetch-execute cycle. At the end of the fetch operation, what is the status of the instruction? Specifically, what has the fetch operation achieved that prepares the instruction for execution? Explain the similarity between this operation and the corresponding operation performed steps performed by the Little Man.

7.11 Once the fetch operation is complete, what is the first step of the execution phase for any instruction that accesses a memory address for data (e.g., LOAD, STORE)?

7.12 Using the ADD instruction as a model, show the fetch-execute cycle for a SUBTRACT instruction.

7.13 Assume the following values in various registers and memory locations at a given point in time:
PC: 20 A: 150 Memory location 20: 160 [ADD 60] Memory location 60: 30.
Show the values that are stored in each of the following registers at the completion of the instruction: PC, MAR, MDR, IR, and A.

7.14 Define a *bus*. What are buses used for?

7.15 What three types of "data" might a bus carry?

7.16 Explain how data travels on a bus when the bus is *simplex. Half-duplex. Full-duplex*.

7.17 What is the difference between a *multipoint* bus and a *point-to-point* bus? Draw diagrams that illustrate the difference.

7.18 Briefly describe each of the major disadvantages of parallel buses.

7.19 Which Little Man Computer instructions would be classified as *data movement* instructions?

7.20 What operations would you expect the arithmetic class of instructions to perform?

7.21 Explain the difference between SHIFT and ROTATE instructions.

7.22 What do *program control instructions* do? Which LMC instructions would be classified as program control instructions?

7.23 What is a stack? Explain how a stack works. Create a diagram that shows how PUSH and POP instructions are used to implement a stack.

7.24 What is a *privileged instruction*? Which LMC instructions would normally be privileged?

7.25 Show a 32-bit instruction format that allows 32 different op codes. How many bits are available for addressing in your format?

7.26 Show an instruction format that could be used to move data or perform arithmetic between two registers. Assume that the instruction is 32 bits wide and that the computer has sixteen general-purpose data registers. If the op code uses 8 bits, how many bits are spares, available for other purposes, such as special addressing techniques?

EXERCISES

7.1 Draw side-by-side flow diagrams that show how the Little Man executes a store instruction and the corresponding CPU fetch-execute cycle.

7.2 Suppose that the following instructions are found at the given locations in memory:

```
20 LDA 50
21 ADD 51
50 724
51 006
```

a. Show the contents of the IR, the PC, the MAR, the MDR, and A at the conclusion of instruction 20.

b. Show the contents of each register as each step of the fetch-execute cycle is performed for instruction 21.

7.3 One large modern computer has a 48-bit memory address register. How much memory can this computer address?

7.4 Why are there two different registers (MAR and MDR) associated with memory? What are the equivalents in the Little Man Computer?

7.5 Show the steps of the CPU fetch-execute cycle for the remaining instructions in the Little Man instruction set.

7.6 Most of the registers in the machine have two-way copy capability; that is, you can copy to them from another register, and you can copy from them to another register. The MAR, on the other hand, is always used as a destination register; you only copy to the MAR. Explain clearly why this is so.

7.7 **a.** What is the effect of shifting an unsigned number in a register two bits to the left? One bit to the right? Assume that 0s are inserted to replace bit locations at the end of the register that have become empty due to the shift.

b. Suppose the number is signed, that is, stored using 2's complement. Now what is the effect of shifting the number?

c. Suppose that the shift excludes the sign bit, so that the sign bit always remains the same. Furthermore, suppose that during a right shift, the sign bit is always used as the insertion bit at the left end of the number (instead of 0). Now what is the effect of these shifts?

7.8 If you were building a computer to be used in outer space, would you be likely to use some form of flash memory or RAM as main memory? Why?

7.9 Using the register operations indicated in this chapter, show the fetch-execute cycle for an instruction that produces the 2's complement of the number in A. Show the fetch-execute cycle for an instruction that clears A (i.e., sets A to 0).

7.10 Many older computers used an alternative to the BRANCH ON CONDITION instruction called SKIP ON CONDITION that worked as follows: if the condition were true, the computer would skip the following instruction and go on to the one after; otherwise, the next instruction in line would be executed. Programmers usually place a jump instruction in the "in-between" location to branch on a FALSE

condition. Normally, the skip instruction was designed to skip one memory location. If the instruction set uses variable length instructions, however, the task is more difficult, since the skip must still skip around the entire instruction. Assume a Little Man mutant that uses a variable length instruction. The op code is in the first word, and there may be as many as three words following. To make life easy, assume that the third digit of the op code word is a number from 1 to 4, representing the number of words in the instruction. Create a fetch-execute cycle for this machine.

7.11 Suppose that the instruction format for a modified Little Man Computer requires two consecutive locations for each instruction. The high-order digits of the instruction are located in the first mail slot, followed by the low-order digits. The IR is large enough to hold the entire instruction and can be addressed as IR [high] and IR [low] to load it. You may assume that the op code part of the instruction uses IR [high] and that the address is found in IR [low]. Write the fetch-execute cycle for an ADD instruction on this machine.

7.12 The Little Prince Computer (LPC) is a mutant variation on the LMC. (The LPC is so named because the differences are a royal pain.) The LPC has one additional instruction. The extra instruction requires two consecutive words:

```
0XX
0YY
```

This instruction, known as move, moves data directly from location XX to location YY without affecting the value in the accumulator. To execute this instruction, the Little Prince would need to store the XX data temporarily. He can do this by writing the value on a piece of paper and holding it until he retrieves the second address. The equivalent in a real CPU might be called the intermediate address register, or IAR. Write the fetch-execute cycle for the LPC MOVE instruction.

7.13 Generally, the distance that a programmer wants to move from the current instruction location on a BRANCH ON CONDITION is fairly small. This suggests that it might be appropriate to design the BRANCH instruction in such a way that the new location is calculated relative to the current instruction location. For example, we could design a different LMC instruction 8CX. The C digit would specify the condition on which to branch, and X would be a single-digit relative address. Using 10's complement, this would allow a branch of -5 to $+4$ locations from the current address. If we were currently executing this instruction at location 24, 803 would cause a branch on negative to location 27. Write a fetch-execute cycle for this BRANCH ON NEGATIVE RELATIVE instruction. You may ignore the condition code for this exercise, and you may also assume that the complementary addition is handled correctly. The single-digit address, X, is still found in IR [address].

7.14 As computer words get larger and larger, there is a law of diminishing returns: the speed of execution of real application programs does not increase and may, in fact, decrease. Why do you suppose that this is so?

7.15 Most modern computers provide a large number of general-purpose registers and very few memory access instructions. Most instructions use these registers to hold data instead of memory. What are the advantages to such an architecture?

7.16 Create the fetch-execute cycle for an instruction that moves a value from general purpose register-1 to general purpose register-2. Compare this cycle to the cycle for a LOAD instruction. What is the major advantage of the MOVE over the LOAD?

7.17 What are the trade-offs in using a serial bus versus a parallel bus to move data from one place to another?

7.18 Until recently, most personal computers used a parallel PCI bus as a backplane to interconnect the various components within the computer, but the PCI bus was rarely, if ever, used to connect external devices to the computer. Modern computers often use a serial adaptation of the PCI bus called PCI Express, which is sometimes made available as a port to connect external devices. Identify at least three shortcomings of the original PCI bus that made external use of the bus impractical. Explain how the PCI Express bus overcomes each of these limitations.

7.19 Explain why skew is not a factor in a serial bus.

7.20 Point-to-point buses generally omit lines for addressing. Why is this possible? Suppose a point-to-point bus is used to connect two components together where one of the components actually represents multiple addresses. How could a bus with no address lines be used to satisfy the requirement for different addresses in this case?