

Data Manipulation 2

In this chapter we will learn how a computer manipulates data and communicates with peripheral devices such as printers and keyboards. In doing so, we will explore the basics of computer architecture and learn how computers are programmed by means of encoded instructions, called machine language instructions.

2.1 Computer Architecture

CPU Basics

The Stored-Program Concept

2.2 Machine Language

The Instruction Repertoire

An Illustrative Machine Language

2.3 Program Execution

An Example of Program

Execution

Programs Versus Data

*2.4 Arithmetic/Logic Instructions

Logic Operations

Rotation and Shift Operations

Arithmetic Operations

*2.5 Communicating with Other Devices

The Role of Controllers

Direct Memory Access

Handshaking

Popular Communication Media

Communication Rates

*2.6 Other Architectures

Pipelining

Multiprocessor Machines

**Asterisks indicate suggestions for optional sections.*

In Chapter 1 we studied topics relating to the storage of data inside a computer. In this chapter we will see how a computer manipulates that data. This manipulation consists of moving data from one location to another as well as performing operations such as arithmetic calculations, text editing, and image manipulation. We begin by extending our understanding of computer architecture beyond that of data storage systems.

2.1 Computer Architecture

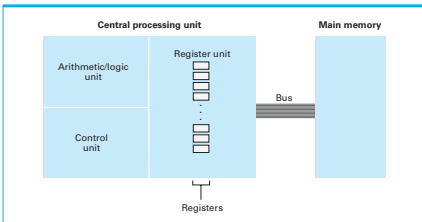
The circuitry in a computer that controls the manipulation of data is called the **central processing unit**, or **CPU** (often referred to as merely the processor). In the machines of the mid-twentieth century, CPUs were large units comprised of perhaps several racks of electronic circuitry that reflected the significance of the unit. However, technology has shrunk these devices drastically. The CPUs found in today's desktop computers and notebooks are packaged as small flat squares (approximately two inches by two inches) whose connecting pins plug into a socket mounted on the machine's main circuit board (called the **motherboard**). In smartphones, mini-notebooks, and other **Mobile Internet Devices (MID)**, CPU's are around half the size of a postage stamp. Due to their small size, these processors are called **microprocessors**.

CPU Basics

A CPU consists of three parts (Figure 2.1): the **arithmetic/logic unit**, which contains the circuitry that performs operations on data (such as addition and subtraction); the **control unit**, which contains the circuitry for coordinating the machine's activities; and the **register unit**, which contains data storage cells (similar to main memory cells), called **registers**, that are used for temporary storage of information within the CPU.

Some of the registers within the register unit are considered **general-purpose registers** whereas others are **special-purpose registers**. We will discuss some of

Figure 2.1 CPU and main memory connected via a bus



the special-purpose registers in Section 2.3. For now, we are concerned only with the general-purpose registers.

General-purpose registers serve as temporary holding places for data being manipulated by the CPU. These registers hold the inputs to the arithmetic/logic unit's circuitry and provide storage space for results produced by that unit. To perform an operation on data stored in main memory, the control unit transfers the data from memory into the general-purpose registers, informs the arithmetic/logic unit which registers hold the data, activates the appropriate circuitry within the arithmetic/logic unit, and tells the arithmetic/logic unit which register should receive the result.

For the purpose of transferring bit patterns, a machine's CPU and main memory are connected by a collection of wires called a **bus** (see again Figure 2.1). Through this bus, the CPU extracts (reads) data from main memory by supplying the address of the pertinent memory cell along with an electronic signal telling the memory circuitry that it is supposed to retrieve the data in the indicated cell. In a similar manner, the CPU places (writes) data in memory by providing the address of the destination cell and the data to be stored together with the appropriate electronic signal telling main memory that it is supposed to store the data being sent to it.

Based on this design, the task of adding two values stored in main memory involves more than the mere execution of the addition operation. The data must be transferred from main memory to registers within the CPU, the values must be added with the result being placed in a register, and the result must then be stored in a memory cell. The entire process is summarized by the five steps listed in Figure 2.2.

The Stored-Program Concept

Early computers were not known for their flexibility—the steps that each device executed were built into the control unit as a part of the machine. To gain more flexibility, some of the early electronic computers were designed so that the CPU could be conveniently rewired. This flexibility was accomplished by means of a pegboard arrangement similar to old telephone switchboards in which the ends of jumper wires were plugged into holes.

Figure 2.2 Adding values stored in memory

- Step 1. Get one of the values to be added from memory and place it in a register.
- Step 2. Get the other value to be added from memory and place it in another register.
- Step 3. Activate the addition circuitry with the registers used in Steps 1 and 2 as inputs and another register designated to hold the result.
- Step 4. Store the result in memory.
- Step 5. Stop.

Cache Memory

It is instructive to compare the memory facilities within a computer in relation to their functionality. Registers are used to hold the data immediately applicable to the operation at hand; main memory is used to hold data that will be needed in the near future; and mass storage is used to hold data that will likely not be needed in the immediate future. Many machines are designed with an additional memory level, called cache memory. **Cache memory** is a portion (perhaps several hundred KB) of high-speed memory located within the CPU itself. In this special memory area, the machine attempts to keep a copy of that portion of main memory that is of current interest. In this setting, data transfers that normally would be made between registers and main memory are made between registers and cache memory. Any changes made to cache memory are then transferred collectively to main memory at a more opportune time. The result is a CPU that can execute its machine cycle more rapidly because it is not delayed by main memory communication.

A breakthrough (credited, apparently incorrectly, to John von Neumann) came with the realization that a program, just like data, can be encoded and stored in main memory. If the control unit is designed to extract the program from memory, decode the instructions, and execute them, the program that the machine follows can be changed merely by changing the contents of the computer's memory instead of rewiring the CPU.

The idea of storing a computer's program in its main memory is called the **stored-program concept** and has become the standard approach used today—so standard, in fact, that it seems obvious. What made it difficult originally was that everyone thought of programs and data as different entities: Data were stored in memory; programs were part of the CPU. The result was a prime example of not seeing the forest for the trees. It is easy to be caught in such ruts, and the development of computer science might still be in many of them today without our knowing it. Indeed, part of the excitement of the science is that new insights are constantly opening doors to new theories and applications.

Questions & Exercises

1. What sequence of events do you think would be required to move the contents of one memory cell in a computer to another memory cell?
2. What information must the CPU supply to the main memory circuitry to write a value into a memory cell?
3. Mass storage, main memory, and general-purpose registers are all storage systems. What is the difference in their use?

2.2 Machine Language

To apply the stored-program concept, CPUs are designed to recognize instructions encoded as bit patterns. This collection of instructions along with the encoding system is called the **machine language**. An instruction expressed in this language is called a machine-level instruction or, more commonly, a **machine instruction**.

The Instruction Repertoire

The list of machine instructions that a typical CPU must be able to decode and execute is quite short. In fact, once a machine can perform certain elementary but well-chosen tasks, adding more features does not increase the machine's theoretical capabilities. In other words, beyond a certain point, additional features may increase such things as convenience but add nothing to the machine's fundamental capabilities.

The degree to which machine designs should take advantage of this fact has led to two philosophies of CPU architecture. One is that a CPU should be designed to execute a minimal set of machine instructions. This approach leads to what is called a **reduced instruction set computer (RISC)**. The argument in favor of RISC architecture is that such a machine is efficient, fast, and less expensive to manufacture. On the other hand, others argue in favor of CPUs with the ability to execute a large number of complex instructions, even though many of them are technically redundant. The result of this approach is known as a **complex instruction set computer (CISC)**. The argument in favor of CISC architecture is that the more complex CPU can better cope with the ever increasing complexities

Who Invented What?

Awarding a single individual credit for an invention is always a dubious undertaking. Thomas Edison is credited with inventing the incandescent lamp, but other researchers were developing similar lamps, and in a sense Edison was lucky to be the one to obtain the patent. The Wright brothers are credited with inventing the airplane, but they were competing with and benefited from the work of many contemporaries, all of whom were preempted to some degree by Leonardo da Vinci, who toyed with the idea of flying machines in the fifteenth century. Even Leonardo's designs were apparently based on earlier ideas. Of course, in these cases the designated inventor still has legitimate claims to the credit bestowed. In other cases, history seems to have awarded credit inappropriately—an example is the stored-program concept. Without a doubt, John von Neumann was a brilliant scientist who deserves credit for numerous contributions. But one of the contributions for which popular history has chosen to credit him, the stored-program concept, was apparently developed by researchers led by J. P. Eckert at the Moore School of Electrical Engineering at the University of Pennsylvania. John von Neumann was merely the first to publish work reporting the idea and thus computer lore has selected him as the inventor.

of today's software. With CISC, programs can exploit a powerful rich set of instructions, many of which would require a multi-instruction sequence in a RISC design.

In the 1990s and into the millennia, commercially available CISC and RISC processors were actively competing for dominance in desktop computing. Intel processors, used in PCs, are examples of CISC architecture; PowerPC processors (developed by an alliance between Apple, IBM, and Motorola) are examples of RISC architecture and were used in the Apple Macintosh. As time progressed, the manufacturing cost of CISC was drastically reduced; thus Intel's processors (or their equivalent from AMD—Advanced Micro Devices, Inc.) are now found in virtually all desktop and laptop computers (even Apple is now building computers based on Intel products).

While CISC secured its place in desktop computers, it has an insatiable thirst for electrical power. In contrast, the company Advanced RISC Machine (ARM) has designed a RISC architecture specifically for low power consumption. (Advanced RISC Machine was originally Acorn Computers and is now ARM Holdings.) Thus, ARM-based processors, manufactured by a host of vendors including Qualcomm and Texas Instruments, are readily found in game controllers, digital TVs, navigation systems, automotive modules, cellular telephones, smartphones, and other consumer electronics.

Regardless of the choice between RISC and CISC, a machine's instructions can be categorized into three groupings: (1) the data transfer group, (2) the arithmetic/logic group, and (3) the control group.

Data Transfer The data transfer group consists of instructions that request the movement of data from one location to another. Steps 1, 2, and 4 in Figure 2.2 fall into this category. We should note that using terms such as *transfer* or *move* to identify this group of instructions is actually a misnomer. It is rare that the data being transferred is erased from its original location. The process involved in a transfer instruction is more like copying the data rather than moving it. Thus terms such as *copy* or *clone* better describe the actions of this group of instructions.

While on the subject of terminology, we should mention that special terms are used when referring to the transfer of data between the CPU and main memory. A request to fill a general-purpose register with the contents of a

Variable-Length Instructions

To simplify explanations in the text, the machine language used for examples in this chapter (and described in Appendix C) uses a fixed size (two bytes) for all instructions. Thus, to fetch an instruction, the CPU always retrieves the contents of two consecutive memory cells and increments its program counter by two. This consistency streamlines the task of fetching instructions and is characteristic of RISC machines. CISC machines, however, have machine languages whose instructions vary in length. Today's Intel processors, for example, have instructions that range from single-byte instructions to multiple-byte instructions whose length depends on the exact use of the instruction. CPUs with such machine languages determine the length of the incoming instruction by the instruction's op-code. That is, the CPU first fetches the op-code of the instruction and then, based on the bit pattern received, knows how many more bytes to fetch from memory to obtain the rest of the instruction.

memory cell is commonly referred to as a LOAD instruction; conversely, a request to transfer the contents of a register to a memory cell is called a STORE instruction. In Figure 2.2, Steps 1 and 2 are LOAD instructions, and Step 4 is a STORE instruction.

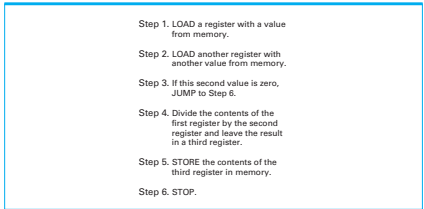
An important group of instructions within the data transfer category consists of the commands for communicating with devices outside the CPU-main memory context (printers, keyboards, display screens, disk drives, etc.). Since these instructions handle the input/output (I/O) activities of the machine, they are called **I/O instructions** and are sometimes considered as a category in their own right. On the other hand, Section 2.5 describes how these I/O activities can be handled by the same instructions that request data transfers between the CPU and main memory. Thus, we shall consider the I/O instructions to be a part of the data transfer group.

Arithmetic/Logic The arithmetic/logic group consists of the instructions that tell the control unit to request an activity within the arithmetic/logic unit. Step 3 in Figure 2.2 falls into this group. As its name suggests, the arithmetic/logic unit is capable of performing operations other than the basic arithmetic operations. Some of these additional operations are the Boolean operations AND, OR, and XOR, introduced in Chapter 1, which we will discuss in more detail later in this chapter.

Another collection of operations available within most arithmetic/logic units allows the contents of registers to be moved to the right or the left within the register. These operations are known as either SHIFT or ROTATE operations, depending on whether the bits that “fall off the end” of the register are merely discarded (SHIFT) or are used to fill the holes left at the other end (ROTATE).

Control The control group consists of those instructions that direct the execution of the program rather than the manipulation of data. Step 5 in Figure 2.2 falls into this category, although it is an extremely elementary example. This group contains many of the more interesting instructions in a machine's repertoire, such as the family of JUMP (or BRANCH) instructions used to direct the CPU to execute an instruction other than the next one in the list. These JUMP instructions appear in two varieties: **unconditional jumps** and **conditional jumps**.

Figure 2.3 Dividing values stored in memory

- 
- Step 1. LOAD a register with a value from memory.
 - Step 2. LOAD another register with another value from memory.
 - Step 3. If this second value is zero, JUMP to Step 6.
 - Step 4. Divide the contents of the first register by the second register and leave the result in a third register.
 - Step 5. STORE the contents of the third register in memory.
 - Step 6. STOP.

An example of the former would be the instruction “Skip to Step 5”; an example of the latter would be, “If the value obtained is 0, then skip to Step 5.” The distinction is that a conditional jump results in a “change of venue” only if a certain condition is satisfied. As an example, the sequence of instructions in Figure 2.3 represents an algorithm for dividing two values where Step 3 is a conditional jump that protects against the possibility of division by zero.

An Illustrative Machine Language

Let us now consider how the instructions of a typical computer are encoded. The machine that we will use for our discussion is described in Appendix C and summarized in Figure 2.4. It has 16 general-purpose registers and 256 main memory cells, each with a capacity of 8 bits. For referencing purposes, we label the registers with the values 0 through 15 and address the memory cells with the values 0 through 255. For convenience we think of these labels and addresses as values represented in base two and compress the resulting bit patterns using hexadecimal notation. Thus, the registers are labeled 0 through F, and the memory cells are addressed 00 through FF.

The encoded version of a machine instruction consists of two parts: the **op-code** (short for operation code) field and the **operand** field. The bit pattern appearing in the op-code field indicates which of the elementary operations, such as STORE, SHIFT, XOR, and JUMP, is requested by the instruction. The bit patterns found in the operand field provide more detailed information about the operation specified by the op-code. For example, in the case of a STORE operation, the information in the operand field indicates which register contains the data to be stored and which memory cell is to receive the data.

The entire machine language of our illustrative machine (Appendix C) consists of only twelve basic instructions. Each of these instructions is encoded using a total of 16 bits, represented by four hexadecimal digits (Figure 2.5). The op-code for each instruction consists of the first 4 bits or, equivalently, the first hexadecimal digit. Note (Appendix C) that these op-codes are represented by the hexadecimal digits 1 through C. In particular, the table in Appendix C shows

Figure 2.4 The architecture of the machine described in Appendix C

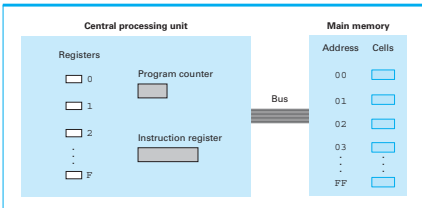
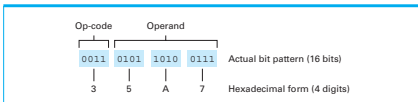
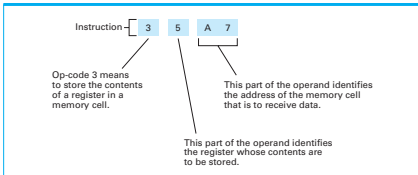


Figure 2.5 The composition of an instruction for the machine in Appendix C

us that an instruction beginning with the hexadecimal digit 3 refers to a STORE instruction, and an instruction beginning with hexadecimal A refers to a ROTATE instruction.

The operand field of each instruction in our illustrative machine consists of three hexadecimal digits (12 bits), and in each case (except for the HALT instruction, which needs no further refinement) clarifies the general instruction given by the op-code. For example (Figure 2.6), if the first hexadecimal digit of an instruction were 3 (the op-code for storing the contents of a register), the next hexadecimal digit of the instruction would indicate which register is to be stored, and the last two hexadecimal digits would indicate which memory cell is to receive the data. Thus the instruction 35A7 (hexadecimal) translates to the statement “STORE the bit pattern found in register 5 in the memory cell whose address is A7.” (Note how the use of hexadecimal notation simplifies our discussion. In reality, the instruction 35A7 is the bit pattern 0011010110100111.)

(The instruction 35A7 also provides an explicit example of why main memory capacities are measured in powers of two. Because 8 bits in the instruction are reserved for specifying the memory cell utilized by this instruction, it is possible to reference exactly 2^8 different memory cells. It behooves us therefore to build main memory with this many cells—addressed from 0 to 255. If main memory had more cells, we would not be able to write instructions that distinguished between them; if main memory had fewer cells, we would be able to write instructions that referenced nonexistent cells.)

Figure 2.6 Decoding the instruction 35A7

As another example of how the operand field is used to clarify the general instruction given by op-code, consider an instruction with the op-code 7 (hexadecimal), which requests that the contents of two registers be ORed. (We will see what it means to OR two registers in Section 2.4. For now we are interested merely in how instructions are encoded.) In this case, the next hexadecimal digit indicates the register in which the result should be placed, while the last two hexadecimal digits indicate which two registers are to be ORed. Thus the instruction 70C5 translates to the statement "OR the contents of register C with the contents of register 5 and leave the result in register 0."

A subtle distinction exists between our machine's two LOAD instructions. Here we see that the op-code 1 (hexadecimal) identifies an instruction that loads a register with the contents of a memory cell, whereas the op-code 2 (hexadecimal) identifies an instruction that loads a register with a particular value. The difference is that the operand field in an instruction of the first type contains an address, whereas in the second type the operand field contains the actual bit pattern to be loaded.

Note that the machine has two ADD instructions: one for adding two's complement representations and one for adding floating-point representations. This distinction is a consequence of the fact that adding bit patterns that represent values encoded in two's complement notation requires different activities within the arithmetic/logic unit from adding values encoded in floating-point notation.

We close this section with Figure 2.7, which contains an encoded version of the instructions in Figure 2.2. We have assumed that the values to be added are stored in two's complement notation at memory addresses 6C and 6D and the sum is to be placed in the memory cell at address 6E.

Figure 2.7 An encoded version of the instructions in Figure 2.2

Encoded instructions	Translation
156C	Load register 5 with the bit pattern found in the memory cell at address 6C.
166D	Load register 6 with the bit pattern found in the memory cell at address 6D.
5056	Add the contents of register 5 and 6 as though they were two's complement representation and leave the result in register 0.
306E	Store the contents of register 0 in the memory cell at address 6E.
C000	Halt.

Questions & Exercises

1. Why might the term *move* be considered an incorrect name for the operation of moving data from one location in a machine to another?
2. In the text, JUMP instructions were expressed by identifying the destination explicitly by stating the name (or step number) of the destination within the JUMP instruction (for example, "Jump to Step 6"). A drawback of this technique is that if an instruction name (number) is later changed, we must be sure to find all jumps to that instruction and change that name also. Describe another way of expressing a JUMP instruction so that the name of the destination is not explicitly stated.
3. Is the instruction "If 0 equals 0, then jump to Step 7" a conditional or unconditional jump? Explain your answer.
4. Write the example program in Figure 2.7 in actual bit patterns.
5. The following are instructions written in the machine language described in Appendix C. Rewrite them in English.
 - a. 368A
 - b. BADE
 - c. 803C
 - d. 40F4
6. What is the difference between the instructions 15AB and 25AB in the machine language of Appendix C?
7. Here are some instructions in English. Translate each of them into the machine language of Appendix C.
 - a. LOAD register number 3 with the hexadecimal value 56.
 - b. ROTATE register number 5 three bits to the right.
 - c. AND the contents of register A with the contents of register 5 and leave the result in register 0.

2.3 Program Execution

A computer follows a program stored in its memory by copying the instructions from memory into the CPU as needed. Once in the CPU, each instruction is decoded and obeyed. The order in which the instructions are fetched from memory corresponds to the order in which the instructions are stored in memory unless otherwise altered by a JUMP instruction.

To understand how the overall execution process takes place, it is necessary to consider two of the special purpose registers within the CPU: the **instruction register** and the **program counter** (see again Figure 2.4). The instruction register is used to hold the instruction being executed. The program counter contains the address of the next instruction to be executed, thereby serving as the machine's way of keeping track of where it is in the program.

The CPU performs its job by continually repeating an algorithm that guides it through a three-step process known as the **machine cycle**. The steps in the

machine cycle are fetch, decode, and execute (Figure 2.8). During the fetch step, the CPU requests that main memory provide it with the instruction that is stored at the address indicated by the program counter. Since each instruction in our machine is two bytes long, this fetch process involves retrieving the contents of two memory cells from main memory. The CPU places the instruction received from memory in its instruction register and then increments the program counter by two so that the counter contains the address of the next instruction stored in memory. Thus the program counter will be ready for the next fetch.

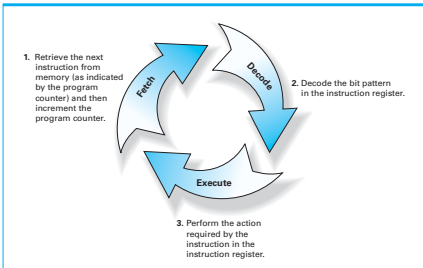
With the instruction now in the instruction register, the CPU decodes the instruction, which involves breaking the operand field into its proper components based on the instruction's op-code.

The CPU then executes the instruction by activating the appropriate circuitry to perform the requested task. For example, if the instruction is a load from memory, the CPU sends the appropriate signals to main memory, waits for main memory to send the data, and then places the data in the requested register; if the instruction is for an arithmetic operation, the CPU activates the appropriate circuitry in the arithmetic/logic unit with the correct registers as inputs and waits for the arithmetic/logic unit to compute the answer and place it in the appropriate register.

Once the instruction in the instruction register has been executed, the CPU again begins the machine cycle with the fetch step. Observe that since the program counter was incremented at the end of the previous fetch, it again provides the CPU with the correct address.

A somewhat special case is the execution of a JUMP instruction. Consider, for example, the instruction B258 (Figure 2.9), which means "JUMP to the instruction at address 58 (hexadecimal) if the contents of register 2 is the same as that of register 0." In this case, the execute step of the machine cycle begins with the comparison of registers 2 and 0. If they contain different bit patterns, the execute step

Figure 2.8 The machine cycle



Comparing Computer Power

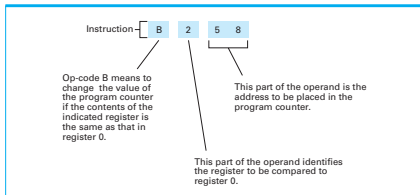
When shopping for a personal computer, you will find that clock speeds are often used to compare machines. A computer's **clock** is a circuit, called an oscillator, which generates pulses that are used to coordinate the machine's activities—the faster this oscillating circuit generates pulses, the faster the machine performs its machine cycle. Clock speeds are measured in hertz (abbreviated as Hz) with one Hz equal to one cycle (or pulse) per second. Typical clock speeds in desktop computers are in the range of a few hundred MHz (older models) to several GHz. (MHz is short for megahertz, which is a million Hz. GHz is short for gigahertz, which is 1000 MHz.)

Unfortunately, different CPU designs might perform different amounts of work in one clock cycle, and thus clock speed alone fails to be relevant in comparing machines with different CPUs. If you are comparing a machine based on an Intel processor to one based on ARM, it would be more meaningful to compare performance by means of **benchmarking**, which is the process of comparing the performance of different machines when executing the same program, known as a benchmark. By selecting benchmarks representing different types of applications, you get meaningful comparisons for various market segments.

terminates and the next machine cycle begins. If, however, the contents of these registers are equal, the machine places the value 58 (hexadecimal) in its program counter during the execute step. In this case, then, the next fetch step finds 58 in the program counter, so the instruction at that address will be the next instruction to be fetched and executed.

Note that if the instruction had been B058, then the decision of whether the program counter should be changed would depend on whether the contents of register 0 was equal to that of register 0. But these are the same registers and thus must have equal content. In turn, any instruction of the form B0XY will cause a jump to be executed to the memory location XY regardless of the contents of register 0.

Figure 2.9 Decoding the instruction B258



An Example of Program Execution

Let us follow the machine cycle applied to the program presented in Figure 2.7, which retrieves two values from main memory, computes their sum, and stores that total in a main memory cell. We first need to put the program somewhere in memory. For our example, suppose the program is stored in consecutive addresses, starting at address A0 (hexadecimal). With the program stored in this manner, we can cause the machine to execute it by placing the address (A0) of the first instruction in the program counter and starting the machine (Figure 2.10).

The CPU begins the fetch step of the machine cycle by extracting the instruction stored in main memory at location A0 and placing this instruction (156C) in its instruction register (Figure 2.11a). Notice that, in our machine, instructions are 16 bits (two bytes) long. Thus the entire instruction to be fetched occupies the memory cells at both address A0 and A1. The CPU is designed to take this into account so it retrieves the contents of both cells and places the bit patterns received in the instruction register, which is 16 bits long. The CPU then adds two to the program counter so that this register contains the address of the next instruction (Figure 2.11b). At the end of the fetch step of the first machine cycle, the program counter and instruction register contain the following data:

Program Counter: A2
Instruction Register: 156C

Next, the CPU analyzes the instruction in its instruction register and concludes that it is to load register 5 with the contents of the memory cell at address 6C. This load activity is performed during the execution step of the machine cycle, and the CPU then begins the next cycle.

This cycle begins by fetching the instruction 166D from the two memory cells starting at address A2. The CPU places this instruction in the instruction

Figure 2.10 The program from Figure 2.7 stored in main memory ready for execution

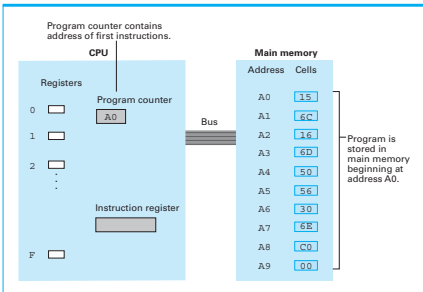
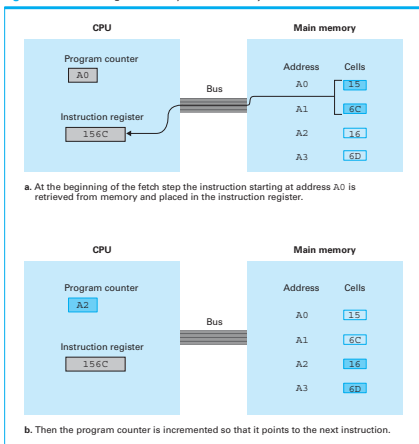


Figure 2.11 Performing the fetch step of the machine cycle



register and increments the program counter to A4. The values in the program counter and instruction register therefore become the following:

Program Counter: A4
Instruction Register: 166D

Now the CPU decodes the instruction 166D and determines that it is to load register 6 with the contents of memory address 6D. It then executes the instruction. It is at this time that register 6 is actually loaded.

Since the program counter now contains A4, the CPU extracts the next instruction starting at this address. The result is that 5056 is placed in the instruction register, and the program counter is incremented to A6. The CPU now decodes the contents of its instruction register and executes it by activating the two's complement addition circuitry with inputs being registers 5 and 6.

During this execution step, the arithmetic/logic unit performs the requested addition, leaves the result in register 0 (as requested by the control unit), and reports to the control unit that it has finished. The CPU then begins another machine cycle. Once again, with the aid of the program counter, it fetches the

next instruction (306E) from the two memory cells starting at memory location A6 and increments the program counter to A8. This instruction is then decoded and executed. At this point, the sum is placed in memory location 6E.

The next instruction is fetched starting from memory location A8, and the program counter is incremented to AA. The contents of the instruction register (C000) are now decoded as the halt instruction. Consequently, the machine stops during the execute step of the machine cycle, and the program is completed.

In summary, we see that the execution of a program stored in memory involves the same process you and I might use if we needed to follow a detailed list of instructions. Whereas we might keep our place by marking the instructions as we perform them, the CPU keeps its place by using the program counter. After determining which instruction to execute next, we would read the instruction and extract its meaning. Then, we would perform the task requested and return to the list for the next instruction in the same manner that the CPU executes the instruction in its instruction register and then continues with another fetch.

Programs Versus Data

Many programs can be stored simultaneously in a computer's main memory, as long as they occupy different locations. Which program will be run when the machine is started can then be determined merely by setting the program counter appropriately.

One must keep in mind, however, that because data are also contained in main memory and encoded in terms of 0s and 1s, the machine alone has no way of knowing what is data and what is program. If the program counter were assigned the address of data instead of the address of the desired program, the CPU, not knowing any better, would extract the data bit patterns as though they were instructions and execute them. The final result would depend on the data involved.

We should not conclude, however, that providing programs and data with a common appearance in a machine's memory is bad. In fact, it has proved a useful attribute because it allows one program to manipulate other programs (or even itself) the same as it would data. Imagine, for example, a program that modifies itself in response to its interaction with its environment and thus exhibits the ability to learn, or perhaps a program that writes and executes other programs in order to solve problems presented to it.

Questions & Exercises

1. Suppose the memory cells from addresses 00 to 05 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
00	14
01	02
02	34
03	17
04	C0
05	00

If we start the machine with its program counter containing 00, what bit pattern is in the memory cell whose address is hexadecimal 17 when the machine halts?

2. Suppose the memory cells at addresses B0 to B8 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
B0	13
B1	B8
B2	A3
B3	02
B4	33
B5	B8
B6	C0
B7	00
B8	0F

- a. If the program counter starts at B0, what bit pattern is in register number 3 after the first instruction has been executed?
- b. What bit pattern is in memory cell B8 when the halt instruction is executed?
3. Suppose the memory cells at addresses A4 to B1 in the machine described in Appendix C contain the (hexadecimal) bit patterns given in the following table:

Address	Contents
A4	20
A5	00
A6	21
A7	03
A8	22
A9	01
AA	B1
AB	B0
AC	50
AD	02
AE	B0
AF	AA
B0	C0
B1	00

When answering the following questions, assume that the machine is started with its program counter containing A4.

- a. What is in register 0 the first time the instruction at address AA is executed?
- b. What is in register 0 the second time the instruction at address AA is executed?
- c. How many times is the instruction at address AA executed before the machine halts?

4. Suppose the memory cells at addresses F0 to F9 in the machine described in Appendix C contain the (hexadecimal) bit patterns described in the following table:

Address	Contents
F0	20
F1	C0
F2	30
F3	F8
F4	20
F5	00
F6	30
F7	F9
F8	FF
F9	FF

If we start the machine with its program counter containing F0, what does the machine do when it reaches the instruction at address F8?

2.4 Arithmetic/Logic Instructions

As indicated earlier, the arithmetic/logic group of instructions consists of instructions requesting arithmetic, logic, and shift operations. In this section, we look at these operations more closely.

Logic Operations

We introduced the logic operations AND, OR, and XOR (exclusive or) in Chapter 1 as operations that combine two input bits to produce a single output bit. These operations can be extended to operations that combine two strings of bits to produce a single output string by applying the basic operation to individual columns. For example, the result of ANDing the patterns 10011010 and 11001001 results in

```

      10011010
    AND 11001001
    -----
      10001000
  
```

where we have merely written the result of ANDing the 2 bits in each column at the bottom of the column. Likewise, ORing and XORing these patterns would produce

```

      10011010      10011010
    OR 11001001    XOR 11001001
    -----
      11011011      01010011
  
```

One of the major uses of the AND operation is for placing 0s in one part of a bit pattern while not disturbing the other part. Consider, for example, what happens if the byte 00001111 is the first operand of an AND operation. Without knowing the contents of the second operand, we still can conclude that the four most significant bits of the result will be 0s. Moreover, the four least significant bits of