# 5

# Register Transfer and Microoperations

CHAPTER OUTLINE



## 5.1 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called microoperations. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. Some of the digital components introduced in Chap. 3 are registers that implement microoperations. For example, a counter with parallel load is capable of performing the microoperations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

*microoperation*

The internal hardware organization of a digital computer is best defined by specifying:

1. The set of registers it contains and their function.
2. The sequence of microoperations performed on the binary information stored in the registers.
3. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involves a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

*register transfer language*

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this chapter will be utilized in subsequent chapters to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

## 5.2 Register Transfer

*register*

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address

register and is designated by the name *MAR*. Other designations for registers are *PC* (for program counter), *IR* (for instruction register, and *R*1 (for processor register). The individual flip-flops in an *n*-bit register are numbered in sequence from 0 through $n - 1$, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 5.1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 5.1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is *PC*. The symbol *PC*(0-7) or *PC*(*L*) refers to the low-order byte and *PC*(8-15) or *PC*(*H*) to the high-order byte.

*register transfer*

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the content of register *R*1 into register *R*2. It designates a replacement of the content of *R*2 by the content of *R*1. By definition, the content of the source register *R*1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

$$If\ (P = 1)\ then\ (R2 \leftarrow R1)$$

where *P* is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a *control function*. A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

*control function*

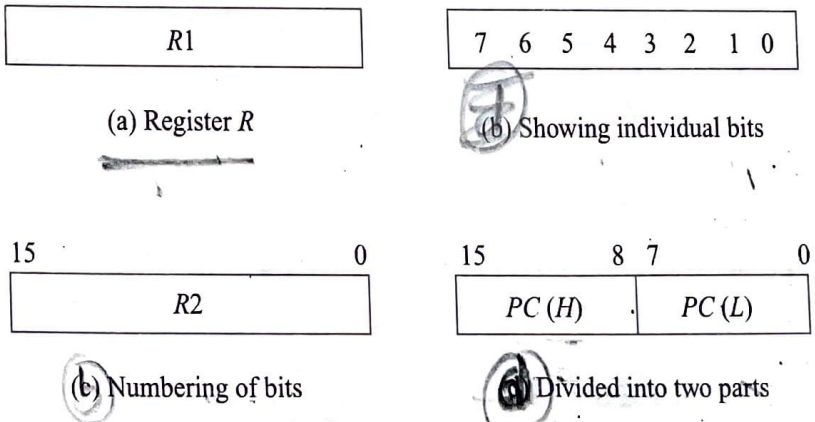



**FIGURE 5.1** Block diagram of register.

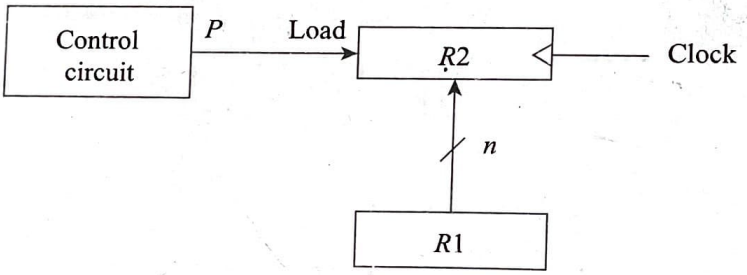
$$P: R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $P = 1$.

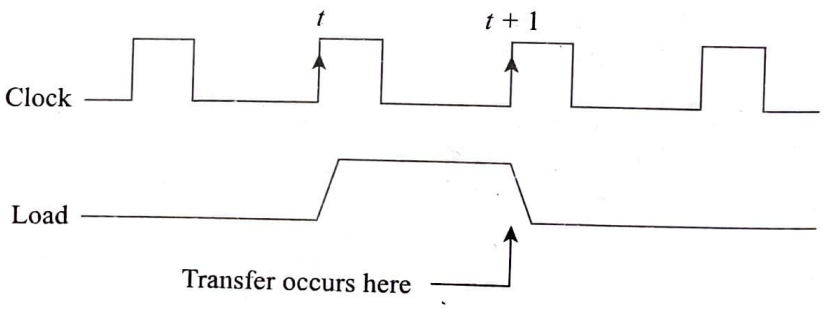
Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 5.2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The $n$ outputs of register $R1$ are connected to the $n$ inputs of register $R2$. The letter $n$ will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load input that is activated by the control variable $P$. It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, $P$ is activated in the control section by the rising edge of a clock pulse at time $t$. The next positive transition of the clock at time $t+1$ finds the load input active and the data inputs of $R2$ are then loaded into the register in parallel. $P$ may go back to 0 at time $t + 1$; otherwise, the transfer will occur with every clock pulse transition while $P$ remains active.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as $P$ becomes active just after time $t$, the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time $t + 1$.

The basic symbols of the register transfer notation are listed in Table 5.1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying




(a) Block diagram




(b) Timing diagram

**FIGURE 5.2** Transfer from $R1$ to $R2$ when $P = 1$.

**TABLE 5.1** Basic Symbols for Register Transfers

| Symbol | Description | Examples |
|---|---|---|
| Letters (and numerals) | Denotes a register | *MAR*, *R2* |
| Parentheses ( ) | Denotes a part of a register | *R2*(0-7), *R2*(*L*) |
| Arrow ← | Denotes transfer of information | *R2* ← *R1* |
| Comma , | Separates two microoperations | *R2* ← *R1*, *R1* ← *R2* |

the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that $T = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

## 5.3 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

*common bus*

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 5.3. Each register has four bits, numbered 0 through 3. The bus consists of four 4 × 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, $S_1$ and $S_0$. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register *A* is connected to input 0 of MUX 1 because this input is labeled $A_1$. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

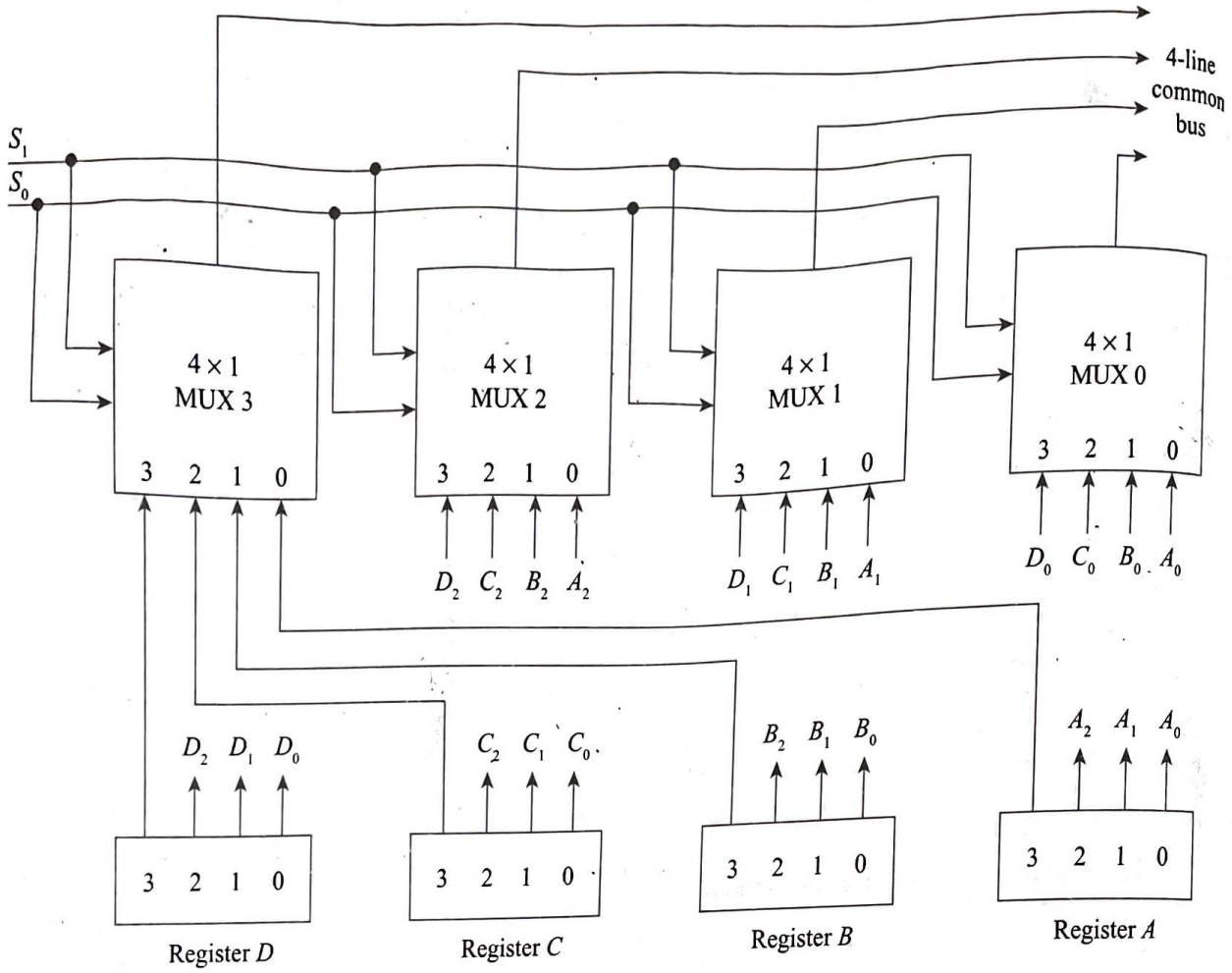



**FIGURE 5.3** Bus system for four registers.

*bus selection*

The two selection lines $S_1$ and $S_0$ are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register $A$ since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register $B$ is selected if $S_1S_0 = 01$, and so on. Table 5.2 shows the register that is selected by the bus for each of the four possible binary value of the selection lines.

In general, a bus system will multiplex $k$ registers of $n$ bits each to produce an $n$-line common bus. The number of multiplexers needed to construct the bus is equal to $n$, the number of bits in each register. The size of

**TABLE 5.2** Function Table for Bus of Fig. 5.3

| $S_1$ | $S_0$ | Register selected |
|---|---|---|
| 0 | 0 | $A$ |
| 0 | 1 | $B$ |
| 1 | 0 | $C$ |
| 1 | 1 | $D$ |

each multiplexer must be $k \times 1$ since it multiplexer $k$ data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is includes in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, \qquad R1 \leftarrow BUS$$

The content of register $C$ is placed on the bus, and the content of the bus is loaded into register $R1$ by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

k - registers.
n - bits (size)
n - line common bus
n - multiplexer
k - size of each multiplexer.
8 - registers of size - 16 bit each
16 - multiplexer one for each line in bus

8 data input lines.
3 selection lines.

## Three-State Bus Buffers

*three-state gate*

*high-impedance buffer*

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a *high-impedance state.* The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in Fig. 5.4. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of

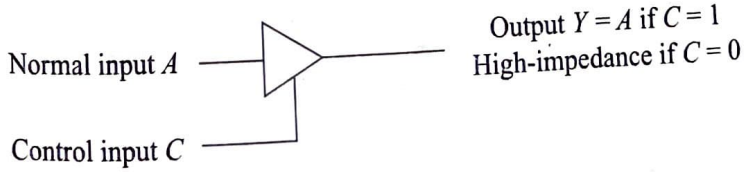



**FIGURE 5.4** Graphic symbols for three-state buffer.

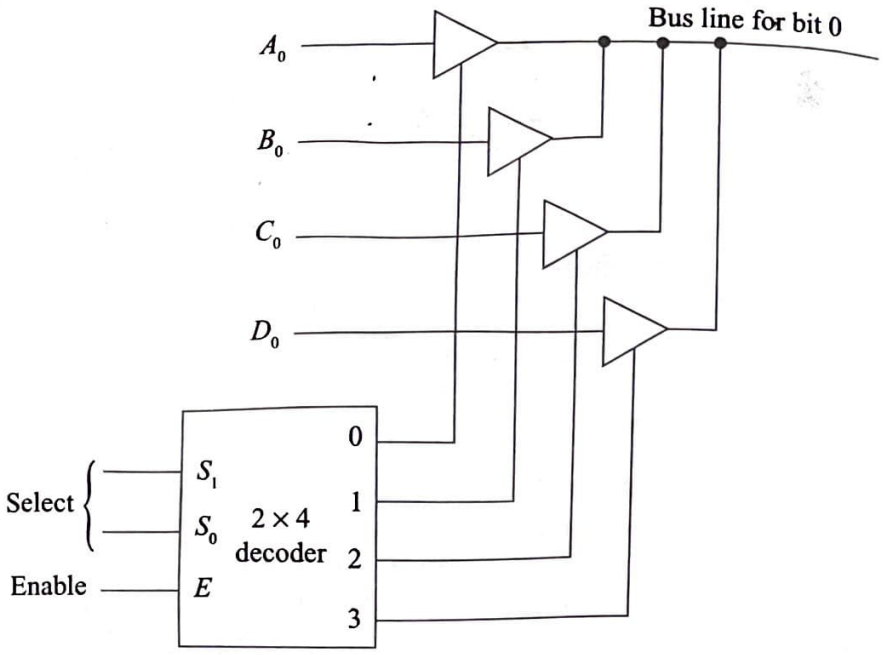



**FIGURE 5.5** Bus line with three state-buffers.

three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

*bus system*

The construction of a bus system with three-state buffers is demonstrated in Fig. 5.5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 5.5 is another way of constructing a $4 \times 1$ multiplexer since the circuit can replace the multiplexer in Fig. 5.3.

To construct a common bus for four registers of $n$ bits each using three-state buffers, we need $n$ circuits with four buffers in each as shown in Fig. 5.5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of $n$ lines. Only one decoder is necessary to select between the four registers.

## Memory Transfer

The operation of a memory unit was described in Sec. 3.7. The transfer of information from a memory word to the outside environment is called a *read*

operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter $M$. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of $M$ when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter $M$.

Consider a memory unit that receives the address from a register, called the address register, symbolized by $AR$. The data are transferred to another register, called the data register, symbolized by $DR$. The read operation can be stated as follows:

*memory read*

$$\text{Read: } DR \leftarrow M[AR]$$

AR → Address Register
DR → Data Register

This causes a transfer of information into $DR$ from the memory word $M$ selected by the address in $AR$.

The write operation transfers the content of a data register to a memory word $M$ selected by the address. Assume that the input data are in register $R1$ and the address is in $AR$. The write operation can be stated symbolically as follows:

*memory write*

$$\text{Write: } M[AR] \leftarrow R1$$

This causes a transfer of information from $R1$ into the memory word $M$ selected by the address in $AR$.

## 5.4 Arithmetic Microoperations

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation was introduced in Sec. 5.2. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperations. In the next two sections we present the logic and shift microoperations.

the organization is arrived through computer design. Thus, computer design is concerned with the hardware design of the computer. Once the architecture and other specifications of the computer are formulated, it is the task of the designer to develop the hardware for the system. Computer design is concerned with the determination of what hardware to be used and how they are to be connected. This aspect of computer hardware is sometimes referred to as *computer implementation*.

## 1.4 Von Neumann Computers

In the very early days of computing, one instruction at a time of a program was executed, with an operator setting up each instruction and also initiating the execution of each instruction. The operator would typically set up an instruction by flipping some switches on a switch board, and then would set up the required data and control paths to the functional units by using patch cords. Certainly, executing a program was extremely cumbersome and required a lot of effort by the operators. At this juncture, in a revolutionary step, the concept of stored program computers was proposed by John Von Neumann in 1945. Von Neumann proposed that instructions can be encoded and stored in the memory just like data. During the execution of a program, the stored instructions can be fetched from memory, and the fetched instruction can be decoded to set up the necessary data paths and generate the control signals.

The organization of a Von Neumann computer is shown schematically in Fig. 1.2. The figure also shows that a control unit is incorporated as a part of the processor. The control unit is an electronic circuitry that decodes each fetched instruction, and generates the necessary control signals that are transmitted to the functional units to carry out the required operations and also the control signals are responsible to automatically set up the required data paths. In essence, the stored program concept along with the control unit, eliminated the need for an operator to enter each instruction through switch settings, and also eliminated the need to set up data paths by inserting patch cords. Von Neumann's novel computer organization not only made program execution much faster, but also set the trend of computer organizations for quite some time to come. Execution of programs became faster as manual entry of the instructions, as well as setting up of the data and control paths to the functional units could be totally avoided. This gave rise to the concept of an *instruction cycle*. In an instruction cycle, an instruction is fetched from memory, analyzed, the data

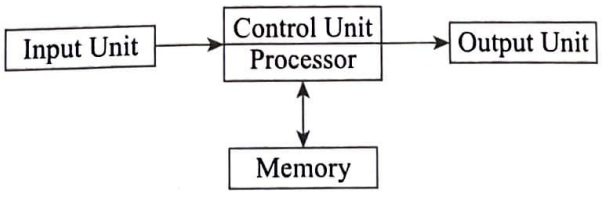



**FIGURE 1.2** Organization of a Von Neumann computer.

fetched, processed, and finally the results are produced. In other words, the execution of an instruction is completed in an instruction cycle.

Von Neumann computer organization was a revolutionary concept and held the center stage in computer design for the next several decades. That is, the organizations of all computers that were designed over the next few decades were essentially based on the Von Neumann style of architecture. However toward the end of 1980s, the quest for faster computations made designers to notice a few shortcomings of the Von Neumann computing. One of the shortcomings of the Von Neumann computers arises due to the fact that a single connection exists between the processor and memory (see Fig. 1.2). Consequently, at any time only one memory access can occur. That is, at a time either an instruction can be fetched or a data item can be accessed. This appeared as a problem in parallel execution of instructions (this issue is discussed in detail in Chapter 10). The term Von Neumann bottleneck is often used to indicate that both fetching an instruction and accessing (reading and writing) data over the same bus from memory by the processor at the same time in a Von Neumann computer is not possible, this is a bottleneck in achieving high-performance computations.

## 1.5 Basic Organization of a Computer

The important parts of a digital computer are the processor, main memory, hard disk (secondary memory), key board, monitor, and peripheral devices. In the simplest organization of a computer, all these different parts of a computer can be connected through a single bus called a backplane bus as shown in Fig. 1.3.

A single bus interconnecting all the components of a computer is usually called a backplane bus. This is so because the single bus can be considered to be a backbone communication medium, to which various components of the computer are attached. Although a backplane bus was used in the early computers, it was later replaced by multiple specialized buses in modern computers for achieving higher performance.

In the following, we briefly discuss the important parts of a Von Neumann computer.

**Processor:** The processor or the central processing unit (CPU) is responsible for fetching an instruction stored in the memory and executing it. It contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control unit. The control unit generates the

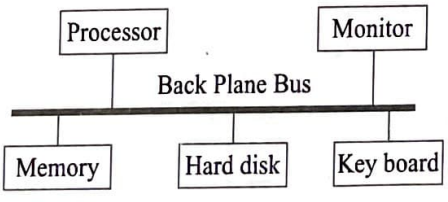



FIGURE 1.3 Basic organization of a digital computer.