CHAPTER 8

Register-Transfer Logic

8.1 Introduction

A digital system is a sequential logic system constructed with flip-flops and gates. It was shown in previous chapters that a sequential circuit can be specified by means of a state table. To specify a large digital system with a state table would be very difficult, if not impossible, because the number of states would be prohibitively large. To overcome this difficulty, digital systems are invariably designed using a modular approach. The system is partitioned into modular subsystems, each of which performs some functional task. The modules are constructed from such digital functions as registers, counters, decoders, multiplexers, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system. A typical digital system module would be the processor unit of a digital computer.

The interconnection of digital functions to form a digital system module cannot be described by means of combinational or sequential logic techniques. These techniques were developed to describe a digital system at the gate and flip-flop level and are not suitable for describing the system at the digital function level. To describe a digital system in terms of functions such as adders, decoders, and registers, it is necessary to employ a higher-level mathematical notation. The register-transfer logic method fulfills this requirement. In this method, the registers are selected to be the primitive components in the digital system, rather than the gates and flip-flops as in sequential logic. In this way it is possible to describe, in a concise and precise manner, the information flow and processing tasks among the data stored in the registers. The register-transfer logic method uses a set of expressions and statements which resemble the statements used in programming languages. This notation provides the necessary tools for specifying a prescribed set of interconnections between various digital functions. An important characteristic of the register-transfer logic method of presentation is that it is closely related to the way people would prefer to specify the operations of a digital system.

The basic components of this method are those that describe a digital system from the operational level. The operation of a digital system is best described by specifying:

1. The set of registers in the system and their functions.
2. The binary-coded information stored in the registers.
3. The operations performed on the information stored in the registers.
4. The control functions that initiate the sequence of operations.

These four components form the basis of the register-transfer logic method for describing digital systems.

A *register,* as defined in the register-transfer logic notation, not only implies a register as defined in Chapter 7, but also encompasses all other types of registers, such as shift registers, counters, and memory units. A counter is considered to be a register whose function is to increment by 1 the information stored within it. A memory unit is considered to be a collection of storage registers where information can be stored. A flip-flop standing alone is taken to be a 1-bit register. In fact, the flip-flops and associated gates of any sequential circuit are called a register by this method of designation.

The *binary information* stored in registers may be binary numbers, binary-coded decimal numbers, alphanumeric characters, control information, or any other binary-coded information. The operations that are performed on the data stored in registers depend on the type of data encountered. Numbers are manipulated with arithmetic operations, whereas control information is usually manipulated with logic operations such as setting and clearing specified bits in the register.

The operations performed on the data stored in registers are called *microoperations.* A microoperation is an elementary operation that can be performed in parallel during one clock pulse period. 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, add, clear, and load. The digital functions introduced in Chapter 7 are registers that implement microoperations. 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. The combinational MSI functions introduced in Chapter 5 can be used in some applications to perform microoperations. A binary parallel adder is useful for implementing the *add* microoperation on the contents of two registers that hold binary numbers. A microoperation requires only one clock pulse for execution if the operation is done in parallel. In serial computers, a microoperation requires a number of pulses equal to the word time in the system. This is equal to the number of bits in the shift registers that transfer the information serially while a microoperation is being executed.

The *control functions* that initiate the sequence of operations consist of timing signals that sequence the operations one at a time. Certain conditions which depend on results of previous operations may also determine the state of control functions. A control function is a binary variable that, when in one binary state, initiates an operation and, when in the other binary state, inhibits the operation.

The purpose of this chapter is to introduce the components of the register-transfer logic method in some detail. The chapter introduces a symbolic notation for representing registers, for specifying operations on the contents of registers, and for specifying control functions. This symbolic notation is sometimes called a *register-transfer language* or *computer hardware description language.* The register-transfer language adopted here is believed to be as simple as possible. It should be realized, however, that no standard symbology exists for a register-transfer language, and different sources adopt different conventions.

A statement in a register-transfer language consists of a control function and a list of microoperations. The control function (which may be omitted sometimes) specifies the control condition and timing sequence for executing the listed micro-operations. The microoperations specify the elementary operations to be performed on the information stored in registers. The

types of microoperations most often encountered in digital systems can be classified into four categories:

1. *Interregister-transfer* microoperations do not change the information content when the binary information moves from one register to another.
2. *Arithmetic* microoperations perform arithmetic on numbers stored in registers.
3. *Logic* microoperations perform operations such as AND and OR on individual pairs of bits stored in registers.
4. *Shift* microoperations specify operations for shift registers.

Sections 8-2 through 8-4 define a basic set of microoperations. Special symbols are assigned to the microoperations in the set, and each symbol is shown to be associated with corresponding digital hardware that implements the stated microoperation. It is important to realize that the register-transfer logic notation is directly related to, and cannot be separated from, the registers and the digital functions that it defines.

The microoperations performed on the information stored in registers depend on the type of data that reside in the registers. The binary information commonly found in registers of digital computers can be classified into three categories:

1. Numerical data such as binary numbers or binary-coded decimal numbers used in arithmetic computations.
2. Nonnumerical data such as alphanumeric characters or other binary-coded symbols used for special applications.
3. Instruction codes, addresses, and other control information used to specify the data-processing requirements in the system.

Sections 8-5 through 8-9 discuss the representation of numerical data and their relationship to the arithmetic microoperations. Section 8-10 explains the use of logic microoperations for processing nonnumerical data. The representation of instruction codes and their manipulation with microoperations are presented in Sections 8-11 and 8-12.

8.2 Interregister Transfer

The registers in a digital system 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 the memory address register and is designated *MAR*. Other designations for registers are *A*, *B*, *R*1, *R*2. and *IR*. The cells or flip-flops of an *n*-bit register are numbered in sequence from 1 to *n* (or from 0 to $n - 1$) starting either from the left or from the right. Figure 8-1 shows four ways to represent a register 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 shown in Fig. 8-1(a). The individual cells can be distinguished as in (b), with each cell assigned a letter with a subscript number. The numbering of cells from right to left can be marked on top of the box as in the 12-bit register *MBR* in (c). A 16-bit register is partitioned into two parts in (d). Bits 1 through 8 are assigned the symbol letter *L* (for low) and bits 9 through 16 are assigned the symbol letter *H* (for high). The name of the 16-bit register is *PC*. The symbol *PC*(*H*) refers to the eight high-ordered cells and *PC*(*L*) refers to the eight low-ordered cells of the register.

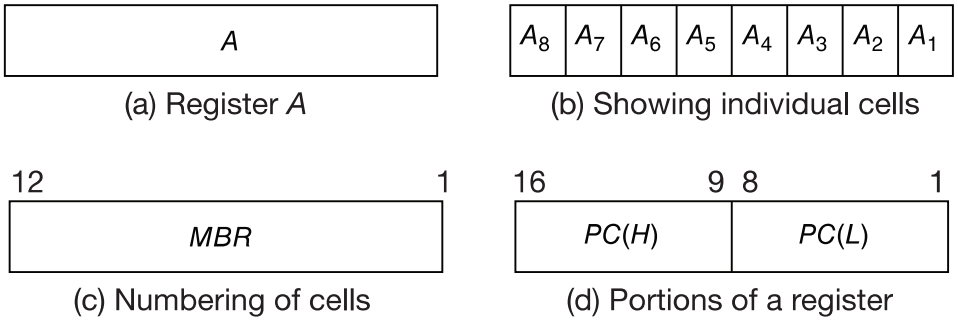

Figure 8.1 Block diagram of registers

Registers can be specified in a register-transfer language with a declaration statement. For example, the registers of Fig. 8-1 can be defined with declaration statements such as:

DECLARE REGISTER $A(8), MBR(12), PC(16)$
DECLARE SUBREGISTER $PC(L) = PC(1{-}8), PC(H) = PC(9{-}16)$

However, in this book we will not use declaration statements to define registers; instead, the registers will be shown in block diagram form as in Fig. 8-1. Registers shown in a block diagram can be easily converted into declaration statements for simulation purposes.

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

$$A \leftarrow B$$

denotes the transfer of the *contents* of register B into register A. It designates a replacement of the contents of A by the contents of B. By definition, the contents of the source register B do 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 cell inputs of the destination register. Normally, we do not want this transfer to occur with every clock pulse, but only under a predetermined condition. The condition that determines when the transfer is to occur is called a *control junction.* A control function is a Boolean function that can be equal to 1 or 0. The control function is included with the statement as follows:

$$x'T_1: A \leftarrow B$$

The control function is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only when the Boolean function $x'T_1 = 1$, i.e., when variable $x = 0$ and timing variable $T_1 = 1$.

Every statement written in a register-transfer language implies a hardware construction for implementing the transfer. Figure 8-2 shows the implementation of the statement written above. The outputs of register B are connected to the inputs of register A, and the number of lines in this connection is equal to the number of bits in the registers. Register A must have a load control input so that it can be enabled when the control function is 1. Although not shown, it is assumed that register A has an additional input that accepts continuous synchronized clock pulses. The control function is generated by means of an inverter and an AND gate. It is also assumed that the

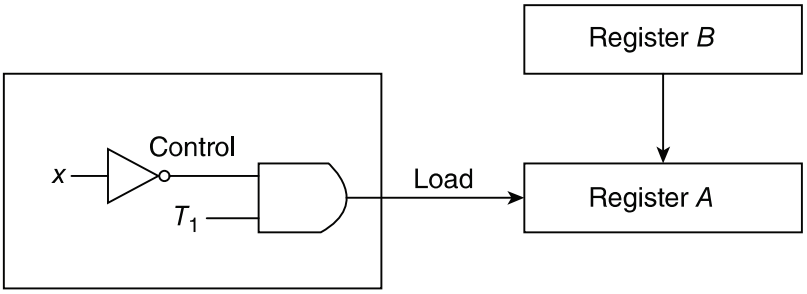

Figure 8.2 Hardware implementation of the statement $x'T_1$: $A \leftarrow B$

control unit that generates the timing variable T_1, is synchronized with the same clock pulses that are applied to register A. The control function stays on during one clock pulse period (when the timing variable is equal to 1), and the transfer occurs during the next transition of a clock pulse.

The basic symbols of the register-transfer logic are listed in Table 8-1. Registers are denoted by capital letters, and numerals may follow the letters. Subscripts are used to distinguish individual cells of the register. Parentheses are used to define a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A colon terminates a control function, and the comma is used to separate two or more operations that are executed at the same time. The statement:

$$T_3: A \leftarrow B, \qquad B \leftarrow A$$

denotes an exchange operation that swaps the contents of two registers during one common clock pulse. This simultaneous operation is possible in registers with master-slave or edge-triggered flip-flops.

The square brackets are used in conjunction with memory transfer. The letter M designates a memory word, and the register enclosed inside the square brackets provides the address for the memory. This is explained in more detail below.

There are occasions when a destination register receives information from two sources, but evidently not at the same time. Consider the two statements:

$$T_1: C \leftarrow A$$
$$T_5: C \leftarrow B$$

Table 8-1 Basic symbol for register-transfer logic

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*A, MBR, R2*
Subscript	Denotes a bit of a register	A_2, B_6
Parentheses ( )	Denotes a portion of a register	*PC*(*H*), *MBR*(*OP*)
Arrow ←	Denotes transfer of information	$A \leftarrow B$
Colon :	Terminates a control function	$x'T_0$:
Comma ,	Separates two microoperations	$A \leftarrow B, B \leftarrow A$
Square brackets [ ]	Specifies an address for memory transfer	$MBR \leftarrow M[MAR]$

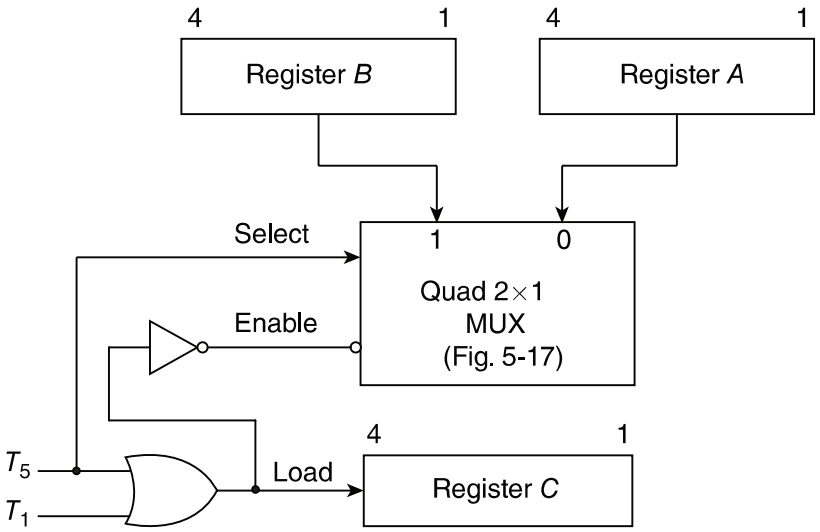

Figure 8.3 Use of a multiplexer to transfer information from two sources into a single destination

The first line states that the contents of register A are to be transferred to register C when timing variable T_1 occurs. The second statement uses the same destination register as the first, but with a different source register and a different timing variable. The connection of two source registers to the same destination register cannot be done directly, but requires a multiplexer circuit to select between two possible paths. The block diagram of the circuit that implements the two statements is shown in Fig. 8-3. For registers with four bits each, we need a quadruple 2-to-l line multiplexer, similar to the one previously given in Fig. 5-17, in order to select either register A or register B. When $T_5 = 1$, register B is selected, but when $T_1 = 1$, register A is selected (because T_5 must be 0 when T_1 is 1). The multiplexer and the load input of register C are enabled every time T_1 or T_5 occurs. This causes a transfer of information from the selected source register into the destination register.

8.2.1 Bus Transfer

Quite often a digital system has many registers, and paths must be provided to transfer information from one register to another register. Consider, for example, the requirement for transfer among three registers as shown in Fig. 8-4. There are six data paths, and each register requires a multiplexer to select between two sources. If each register consists of n flip-flops, there is a need for $6n$ lines and three multiplexers. As the number of registers increases, the number of interconnection lines and multiplexers increases. If we restrict the transfer to one at a time, the number of paths among the registers can be reduced considerably. This is shown in Fig. 8-5, where the

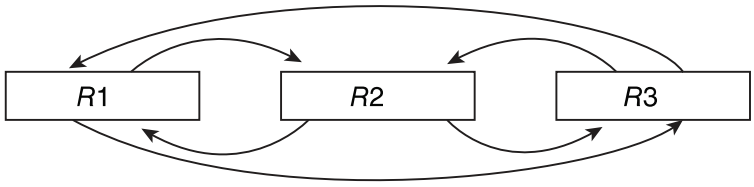

Figure 8.4 Transfer among three registers

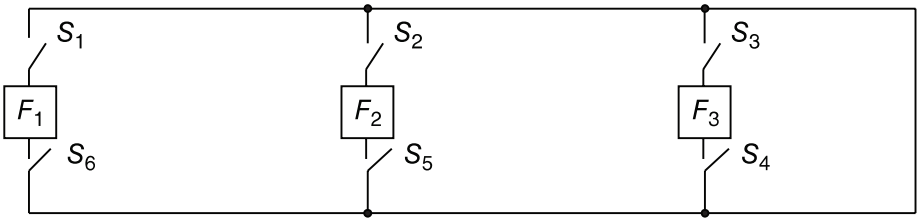

Figure 8.5 Transfer through one common line

output and input of each flip-flop is connected to a common line through an electronic circuit that acts like a switch. All the switches are normally open until a transfer is required. For a transfer from F_1 to F_3, for example, switches S_1 and S_4 are closed to form the required path. This scheme can be extended to registers with n flip-flops, and it requires n common lines.

A group of wires through which binary information is transferred one at a time among registers is called a *bus.* For parallel transfer, the number of lines in the bus is equal to the number of bits in the registers. The idea of a bus transfer is analogous to a central transportation system used to bring commuters from one point to another. Instead of each commuter using private transportation to go from one location to another, a bus system is used, and the commuters wait in line until transportation is available.

A common-bus system can be constructed with multiplexers, and a destination register for the bus transfer can be selected by means of a decoder. The multiplexers select one source register for the bus, and the decoder selects one destination register to transfer the information from the bus. The construction of a bus system for four registers is depicted in Fig. 8-6. The four bits in the same significant position in the registers go through a 4-to-l line multiplexer to form one line of the bus. Only two multiplexers are shown in the diagram: one for the low-order significant bits and one for the high-order significant bits. For registers of n bits, n multiplexers are needed to produce an n-line bus. The n lines in the bus are connected to the n inputs of ail registers. The transfer of information from the bus into one destination register is accomplished by activating the load control of that register. The particular load control activated is selected by the outputs of the decoder when enabled. If the decoder is not enabled, no information will be transferred, even though the multiplexers place the contents of a source register onto the bus.

To illustrate with a particular example, consider the statement:

$$C \leftarrow A$$

The control function that enables this transfer must select register A for the bus and register C for the destination. The multiplexers and decoder select inputs must be:

Select source = 00	(MUXs select register A)
Select destination = 10	(decoder selects register C)
Decoder enable = 0	(decoder is enabled)

On the next clock pulse, the contents of A, being on the bus, are loaded into register C.

8.2.2 Memory Transfer

The operation of a memory unit was described in Section 7-7. The transfer of information from a memory register to the outside environment is called a *read* operation. The transfer of new

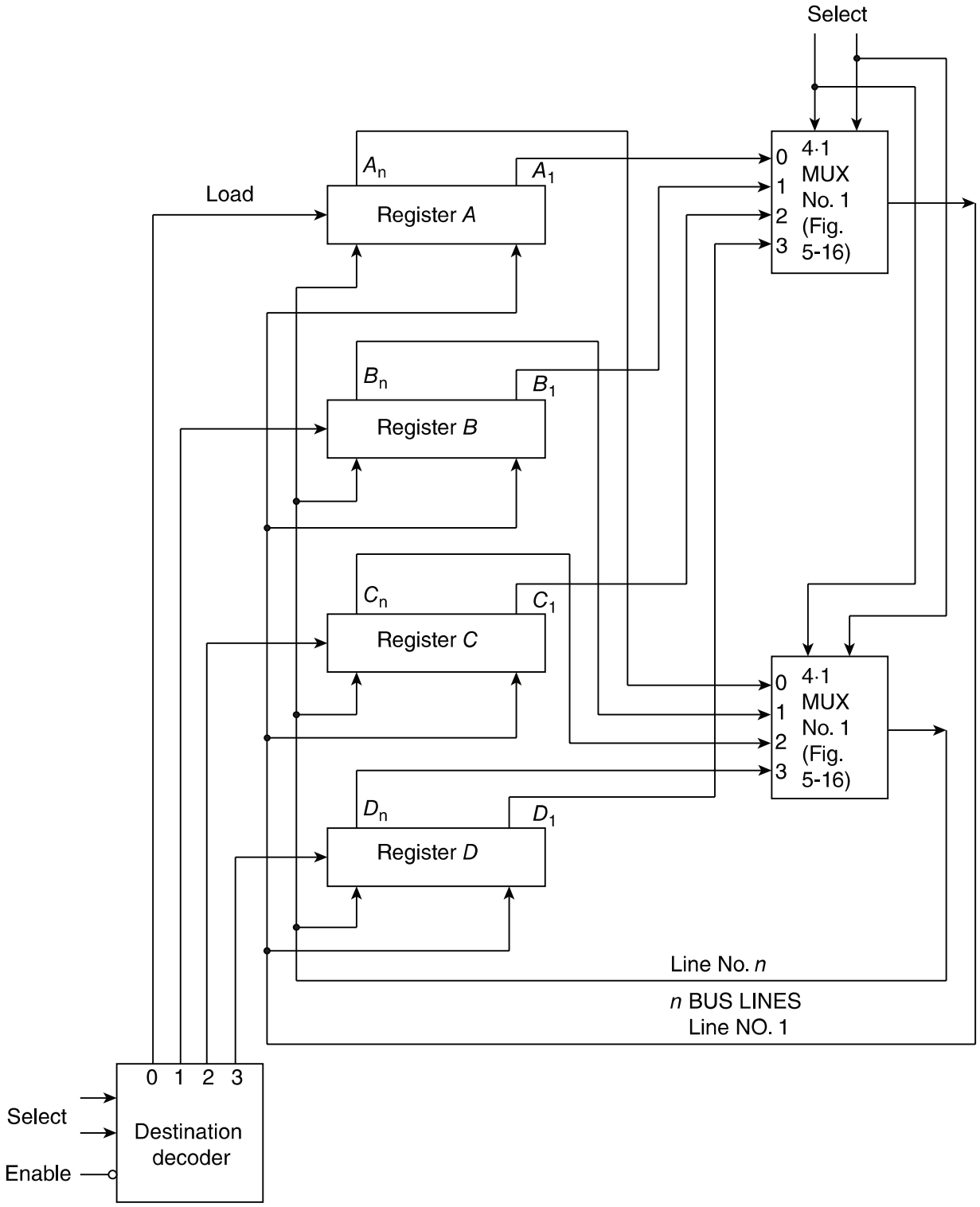

Figure 8.6 Bus system for four registers

information into a memory register is called a *write* operation. In both operations, the memory register selected is specified by an address.

A memory register or word is symbolized by the letter *M.* The particular memory register among the many available in a memory unit is selected by the memory address during the transfer. It is necessary to specify the address of *M* when writing memory-transfer statements. In

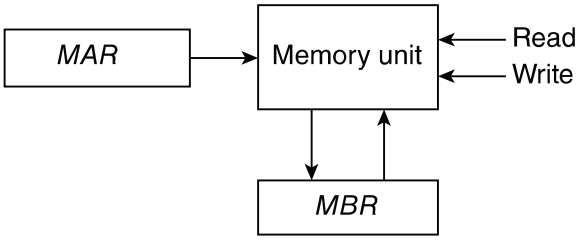

Figure 8.7 Memory unit that communicates with two external registers

some applications, only one address register is connected to the address terminals of the memory. In other applications, the address lines form a common-bus system to allow many registers to specify an address. When only one register is connected to the memory address, we know that this register specifies the address and we can adopt a convention that will simplify the notation. If the letter *M* stands by itself in a statement, it will always designate a memory register selected by the address presently in *MAR*. Otherwise, the register that specifies the address (or the address itself) will be enclosed within square brackets after the symbol *M*.

Consider a memory unit that has a single address register, *MAR*, as shown in Fig. 8-7. The diagram also shows a single memory buffer register *MBR* used to transfer data into and out of memory. There are two memory-transfer operations: read and write. The read operation is a transfer from the selected memory register *M* into *MBR*. This is designated symbolically by the statement:

$$R\text{: } MBR \leftarrow M$$

R is the control function that initiates the read operation. This causes a transfer of information into *MBR* from the selected memory register *M* specified by the address in *MAR*. The write operation is a transfer from *MBR* to the selected memory register *M*. This is designated by the statement:

$$W\text{: } M \leftarrow MBR$$

W is the control function that initiates the write operation. This causes a transfer of information from *MBR* into the memory register *M* selected by the address presently in *MAR*.

The access time of a memory unit must be synchronized with the master clock pulses in the system that triggers the processor registers. In fast memories, the access time may be shorter than or equal to a clock pulse period. In slow memories, it may be necessary to wait for a number of clock pulses for the transfer to be completed. In magnetic-core memories, the processor registers must wait for the memory cycle time to be completed. For a read operation, the cycle time includes the restoration of the word after reading. For a write operation, the cycle time includes the clearing of the memory word prior to writing.

In some systems, the memory unit receives addresses and data from many registers connected to common buses. Consider the case depicted in Fig. 8-8. The address to the memory unit comes from an address bus. Four registers are connected to this bus and any one may supply an address. The output of the memory can go to any one of four registers which are selected by a decoder. The data input to the memory comes from the data bus, which selects one of four registers. A memory word is specified in such a system by the symbol *M* followed by a register enclosed in

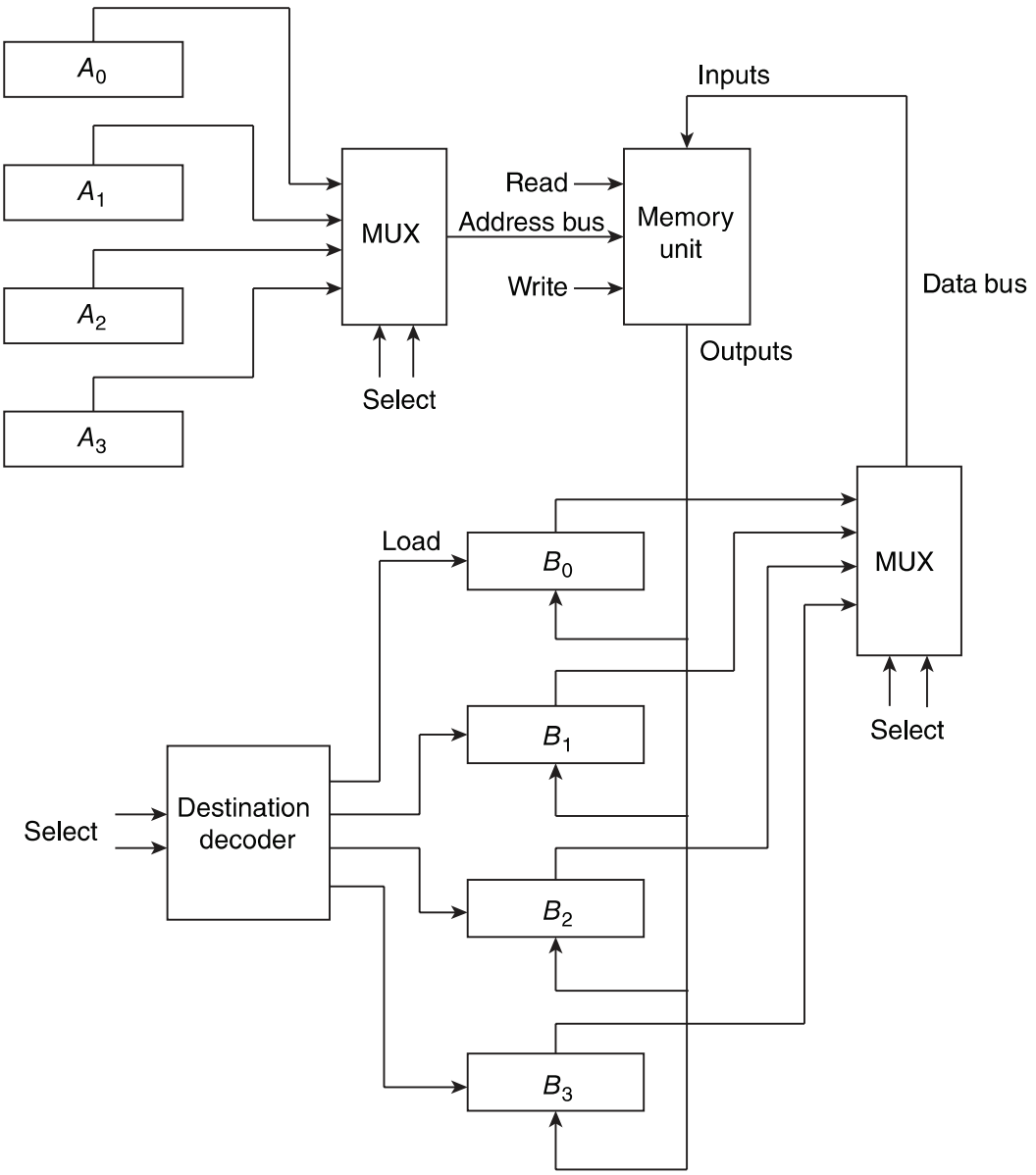

Figure 8.8 Memory unit that communicates with multiple registers

square brackets. The contents of the register within the square brackets specify the address for *M*. The transfer of information from register *B*2 to a memory word selected by the address in register *A*1 is symbolized by the statement:

$$W: M[A1] \leftarrow B2$$

This is a write operation, with register *A*1 specifying the address. The square brackets after the letter *M* give the address register used for selecting the memory register *M*. The statement does not specify the buses explicitly. Nevertheless, it implies the required selection inputs for the two multiplexers that form the address and data buses.

The read operation in a memory with buses can be specified in a similar manner. The statement:

$$R: B0 \leftarrow M[A3]$$

symbolizes a read operation from a memory register whose address is given by $A3$. The binary information coming out of memory is transferred to register $B0$. Again, this statement implies the required selection inputs for the address multiplexer and the selection variables for the destination decoder.

8.3 Arithmetic, Logic, and Shift Microoperations

The interregister-transfer microoperations do not change the information content when the binary information moves from the source register to the destination register. All other microoperations change the information content during the transfer. Among all possible operations that can exist in a digital system, there is a basic set from which all other operations can be obtained. In this section, we define a set of basic microoperations, their symbolic notation, and the digital hardware that implements them. Other microoperations with appropriate symbols can be defined if necessary to suit a particular application.

8.3.1 Arithmetic Microoperations

The basic arithmetic microoperations are add, subtract, complement, and shift. Arithmetic shifts are explained in Section 8-7 in conjunction with the type of binary data representation. All other arithmetic operations can be obtained from a variation or a sequence of these basic microoperations.

The arithmetic microoperation defined by the statement:

$$F \leftarrow A + B$$

specifies an *add* operation. It states that the contents of register A are to be added to the contents of register B, and the sum transferred to register F. To implement this statement, we require three registers, A, B, and F, and the digital function that performs the addition operation, such as a parallel adder. The other basic arithmetic operations are listed in Table 8-2. Arithmetic subtraction implies the availability of a binary parallel subtractor composed of full-subtractor circuits connected in cascade. Subtraction is most often implemented through complementation and addition as specified by the statement:

$$F \leftarrow A + \overline{B} + 1$$

Table 8-2 Arithmetic microoperations

Symbolic designation	Description
$F \leftarrow A + B$	Contents of A plus B transferred to F
$F \leftarrow A - B$	Contents of A minus B transferred to F
$B \leftarrow \overline{B}$	Complement register B (1's complement)
$B \leftarrow \overline{B} + 1$	Form the 2's complement of the contents of register B
$F \leftarrow A + \overline{B} + 1$	A plus the 2's complement of B transferred to F
$A \leftarrow A + 1$	Increment the contents of A by 1 (count up)
$A \leftarrow A - 1$	Decrement the contents of A by 1 (count down)

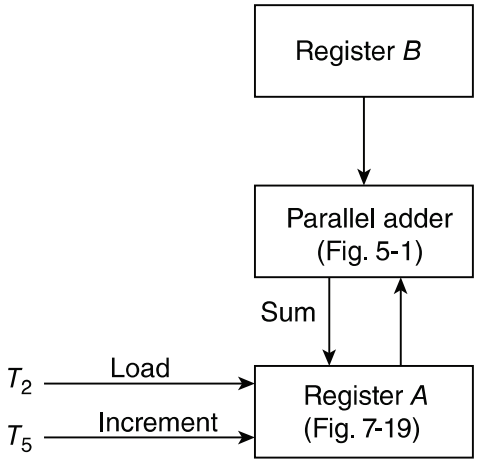

Figure 8.9 Implementation for the add and increment microoperations

$\overline{B}$ is the symbol for the 1's complement of B. Adding 1 to the 1's complement gives the 2's complement of B. Adding A to the 2's complement of B produces A minus B.

The increment and decrement microoperations are symbolized by a *plus-one* or *minus-one* operation executed on the contents of a register. These microoperations are implemented with an up-counter or down-counter, respectively.

There must be a direct relationship between the statements written in a register-transfer language and the registers and digital functions which are required for their implementation. To illustrate this relationship, consider the two statements:

$$T_2\colon A \leftarrow A + B$$
$$T_5\, A \leftarrow A + 1$$

Timing variable T_2 initiates an operation to add the contents of register B to the present contents of A. Timing variable T_5 increments register A. The incrementing can be easily done with a counter, and the sum of two binary numbers can be generated with a parallel adder. The transfer of the sum from the parallel adder into register A can be activated with a load input in the register. This dictates that the register be a counter with parallel-load capability. The implementation of the above two statements is shown in block diagram form in Fig. 8-9. A parallel adder receives input information from registers A and B. The sum bits from the parallel adder are applied to the inputs of A, and timing variable T_2 loads the sum into register A. Timing variable T_5 increments the register by enabling the increment input (or count input, as in Fig. 7-19).

Note that the arithmetic operations multiply and divide are not listed in Table 8-2. The multiplication operation can be represented by the symbol *, and the division by a /. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system where they are implemented by means of combinational circuits. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signals progagate through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware implementation in such a case requires a list of statements that use the basic microoperations of *add, subtract,* and *shift*.

8.3.2 Logic Microoperations

Logic microoperations specify binary operations for a string of bits stored in registers. These operations consider each bit in the registers separately and treat it as a binary variable. As an illustration, the exclusive-OR microoperation is symbolized by the statement:

$$F \leftarrow A \oplus B$$

It specifies a logic operation that considers each pair of bits in the registers as binary variables. If the content of register A is 1010 and that of register B 1100, the information transferred to register F is 0110:

1010	content of A
1100	content of B
0110	content of $F \leftarrow A \oplus B$

There are 16 different possible logic operations that can be performed with two binary variables. These logic operations are listed in Table 2-6. All 16 logic operations can be expressed ill terms of the AND, OR, and complement operations. Special symbols will be adopted for these three microoperations to distinguish them from the corresponding symbols used to express Boolean functions. The symbol $\vee$ will be used to denote an OR microoperation and the symbol $\wedge$ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the letter (or letters) that denotes the register. By using these symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. The symbols for the four logic microoperations are summarized in Table 8-3. The last two symbols are for the shift microoperations discussed below.

A more important reason for adopting a special symbol for the OR microoperation is to differentiate the symbol +, when used as an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When this symbol occurs in a microoperation, it denotes an arithmetic plus. When it occurs in a control (or Boolean) function, it denotes a logic OR operation. For example, in the statement:

$$T_1 + T_2: A \leftarrow A + B,\ C \leftarrow D \vee F$$

Table 8-3 Logic and shift microoperations

Symbolic designation	Description
$A \leftarrow \overline{A}$	Complement all bits of register A
$F \leftarrow A \vee B$	Logic OR microoperation
$F \leftarrow A \wedge B$	Logic AND microoperation
$F \leftarrow A \oplus B$	Logic exclusive-OR microoperation
$A \leftarrow \text{shl } A$	Shift-left register A
$A \leftarrow \text{shr } A$	Shift-right register A

the + between T_1 and T_2 is an OR operation between two timing variables of a control function. The + between A and B specifies an add microoperation. The OR microoperation is designated by the symbol $\vee$ between registers D and F.

The logic microoperations can be easily implemented with a group of gates. The complement of a register of n bits is obtained from n inverter gates. The AND microoperation is obtained from a group of AND gates, each of which receives a pair of bits from the two source registers. The outputs of the AND gates are applied to the inputs of the destination register. The OR microoperation requires a group of OR gates arranged in a similar fashion.

8.3.3 Shift Microoperations

Shift microoperations transfer binary information between registers in serial computers. They are also used in parallel computers for arithmetic, logic, and control operations. Registers can be shifted to the left or to the right. There are no conventional symbols for the shift operations. In this book, we adopt the symbols *shl* and *shr* for the shift-left and shift-right operations, respectively, For example:

$$A \leftarrow \text{shl } A, \qquad B \leftarrow \text{shr } B$$

are two microoperations that specify a l-bit shift to the left of register A and a 1-bit shift to the right of register B. The register symbol must be the same on both sides of the arrow as in the increment operation.

While the bits of a register are shifted, the extreme flip-flops receive information from the serial input. The extreme flip-flop is in the leftmost position of the register during a shift-right operation and in the rightmost position during a shift-left operation. The information transferred into the extreme flip-flops is not specified by the *shl* and *shr* symbols. Therefore, a shift microoperation statement must be accompanied with another microoperation that specifies the value of the serial input for the bit transfer into the extreme flip-flop. For example:

$$A \leftarrow \text{shl } A, \; A_1 \leftarrow A_n$$

is a circular shift chat transfers the leftmost bit from A_n into the rightmost flip-flop A_1 Similarly:

$$A \leftarrow \text{shr } A, \; A_n \leftarrow E$$

is a shift-right operation with the leftmost flip-flop A_n receiving the value of the 1-bit register E.

8.4 Conditional Control Statements

It is sometimes convenient to specify a control condition by a conditional statement rather than a Boolean control function. A *conditional control* statement is symbolized by an *if-then-else* statement in the following manner:

P: If (condition) *then* [microoperation(s)] *else* [microoperation(s)]

The statement is interpreted to mean that if the control condition stated within the parentheses after the word *if* is true, then the microoperation (or microoperations) enclosed within the parentheses after the word *then* is executed. If the condition is not true, the microoperation listed after

the word *else* is executed. In any case, the control function P must occur for anything to be done. If the *else* part of the statement is missing, then nothing is executed if the condition is not true.

The conditional control statement is more of a convenience than a necessity. It enables the writing of clearer statements that are easier for people to interpret. It can always be rewritten in a conventional statement without an if-then-else form. As an example, consider the conditional control statement:

$$T_2\text{: If } (C = 0) \text{ then } (F \leftarrow 1) \text{ else } (F \leftarrow 0)$$

F is assumed to be a 1-bit register (flip-flop) that can be set or cleared. If register C is a 1-bit register, the statement is equivalent to the following two statements:

$$C'T_2\text{: } F \leftarrow 1$$
$$CT_2\text{: } F \leftarrow 0$$

Note that the same timing variable can occur in two separate control functions. The variable C can be either 0 or 1; therefore, only one of the microoperations will be executed during T_2, depending on the value of C.

If register C has more than one bit, the condition $C = 0$ means that all bits of C must be 0. Assume that register C has four bits C_1, C_2, C_3, and C_4. The condition for $C = 0$ can be expressed with a Boolean function:

$$x = C'_1C'_2C'_3C'_4 = (C_1 + C_2 + C_3 + C_4)'$$

Variable x can be generated with a NOR gate. Using the definition of x as above, the conditional control statement is now equivalent to the two statements;

$$xT_2\text{:} \qquad F \leftarrow 1$$
$$x'T_2\text{:} \qquad F \leftarrow 0$$

Variable $x = 1$ if $C = 0$ but is equal to 0 if $C \neq 0$.

When writing conditional control statements, one must realize that the condition stated after the word *if* is part of the control function and not part of a microoperation statement. The condition must be clearly stated and must be implementable with a combinational circuit,

8.5 Fixed-point Binary Data

The binary information found in registers represents either data or control information. Data are operands and other discrete elements of information operated on to achieve required results. Control information is a bit or group of bits that specify the operations to be done. A unit of control information stored in digital computer registers is called an *instruction* and is a binary code that specifies the operations to be performed on the stored data. Instruction codes and their representation in registers are presented in Section 8-11. Some commonly used types of data and their representation in registers are presented in this and the following sections.

8.5.1 Sign and Radix-Point Representation

A register with n flip-flops can store a binary number of n bits; each flip-flop represents one binary digit. This represents the magnitude of the number but does not give information about its