CHAPTER 9

Processor Logic Design

9.1 Introduction

A processor unit is that part of a digital system or a digital computer that implements the operations in the system. It is comprised of a number of registers and the digital functions that implement arithmetic, logic, shift, and transfer microoperations. The processor unit, when combined with a control unit that supervises the sequence of microoperations, is called a *central processor unit* or CPU. This chapter is concerned with the organization and design of the processor unit. The next chapter deals with the logic design of the control unit. In Chapter 11, we demonstrate the organization and design of a computer CPU.

The number of registers in a processor unit may vary from just one processor register to as many as 64 registers or more. Some older computers came with only one processor register. In some cases a special-purpose digital system may employ a single processor register. However, since registers and other digital functions are inexpensive when constructed with integrated circuits, all recent computers employ a large number of processor registers and route the information among them through common buses.

An operation may be implemented in a processor unit either with a single microoperation or with a sequence of microoperations. For example, the multiplication of two binary numbers stored in two registers may be implemented with a combinational circuit that performs the operation by means of gates. As soon as the signals propagate through the gates, the product is available and can be transferred to a destination register with a single clock pulse. Alternatively, the multiplication operation may be performed with a sequence of add and shift microoperations. The method chosen for the implementation dictates the amount and type of hardware in the processor unit.

All computers, except the very large and fast ones, implement the involved operations by means of a sequence of microoperations. In this way, the processor unit need only have circuits that implement simple, basic microoperations such as add and shift. Other operations, such as multiplication, division, and floating-point arithmetic, are generated in conjunction with the control unit. The processor unit by itself is designed to implement basic microoperations of the type discussed in Chapter 8. The control unit is designed to sequence the microoperations to achieve other operations which are not included in the basic set.

The digital function that implements the microoperations on the information stored in processor registers is commonly called an *arithmetic logic unit* or ALU. To perform a microoperation, the control routes the source information from registers into the inputs of the ALU. The

ALU receives the information from the registers and performs a given operation as specified by the control. The result of the operation is then transferred to a destination register. By definition, the ALU is a combinational circuit; thus the entire register-transfer operation can be performed during one clock pulse interval. All register-transfer operations, including interregister transfers, in a typical processor unit are performed in one common ALU; otherwise, it would be necessary to duplicate the digital functions for each register. The shift microoperations are often performed in a separate unit. The shift unit is usually shown separately, but sometimes this unit is implied to be part of the overall arithmetic and logic unit.

A computer CPU must manipulate not only data but also instruction codes and addresses coming from memory. The register that holds and manipulates the operation code of instructions is considered to be part of the control unit. The registers that hold addresses are sometimes included as part of the processor unit, and the address information is manipulated by the common ALU. In some computers, the registers that hold addresses are connected to a separate bus and the address information is manipulated with separate digital functions.

This chapter presents a few alternatives for the organization and design of a processor unit. The design of a particular arithmetic logic unit is undertaken to show the design process involved in formulating and implementing a common digital function capable of performing a large number of microoperations. Other digital functions considered and designed in this chapter are a shifter unit and a general-purpose processor register, commonly called an *accumulator.*

9.2 Processor Organization

The processor part of a computer CPU is sometimes referred to as the *data path* of the CPU because the processor forms the paths for the data transfers between the registers in the unit. The various paths are said to be controlled by means of *gates* that open the required path and close all others. A processor unit can be designed to fulfill the requirements of a set of data paths for a specific application. The design of a special-purpose processor was demonstrated in Section 8-12. Figure 8-16 showed the various data paths for a particular, very limited processor. The gating of the data paths is achieved through the decoders and combinational circuit which comprise the control section of the unit.

In a well-organized processor unit, the data paths are formed by means of buses and other common lines. The control gates that formulate the given path are essentially multiplexers and decoders whose selection lines specify the required path. The processing of information is done by one common digital function whose data path can be specified with a set of common selection variables. A processor unit that has a well-structured organization can be used in a wide variety of applications. If constructed within an integrated circuit, it becomes available to many users, each of which may have a different application.

In this section, we investigate a few alternatives for organizing a general-purpose processor unit. All organizations employ a common ALU and shifter. The differences in organizations are mostly manifested in the organization of the registers and their common path to the ALU.

9.2.1 Bus Organization

When a large number of registers are included in a processor unit, it is most efficient to connect them through common buses or arrange them as a small memory having very fast access time. The registers communicate with each other not only for direct data transfers, but also while

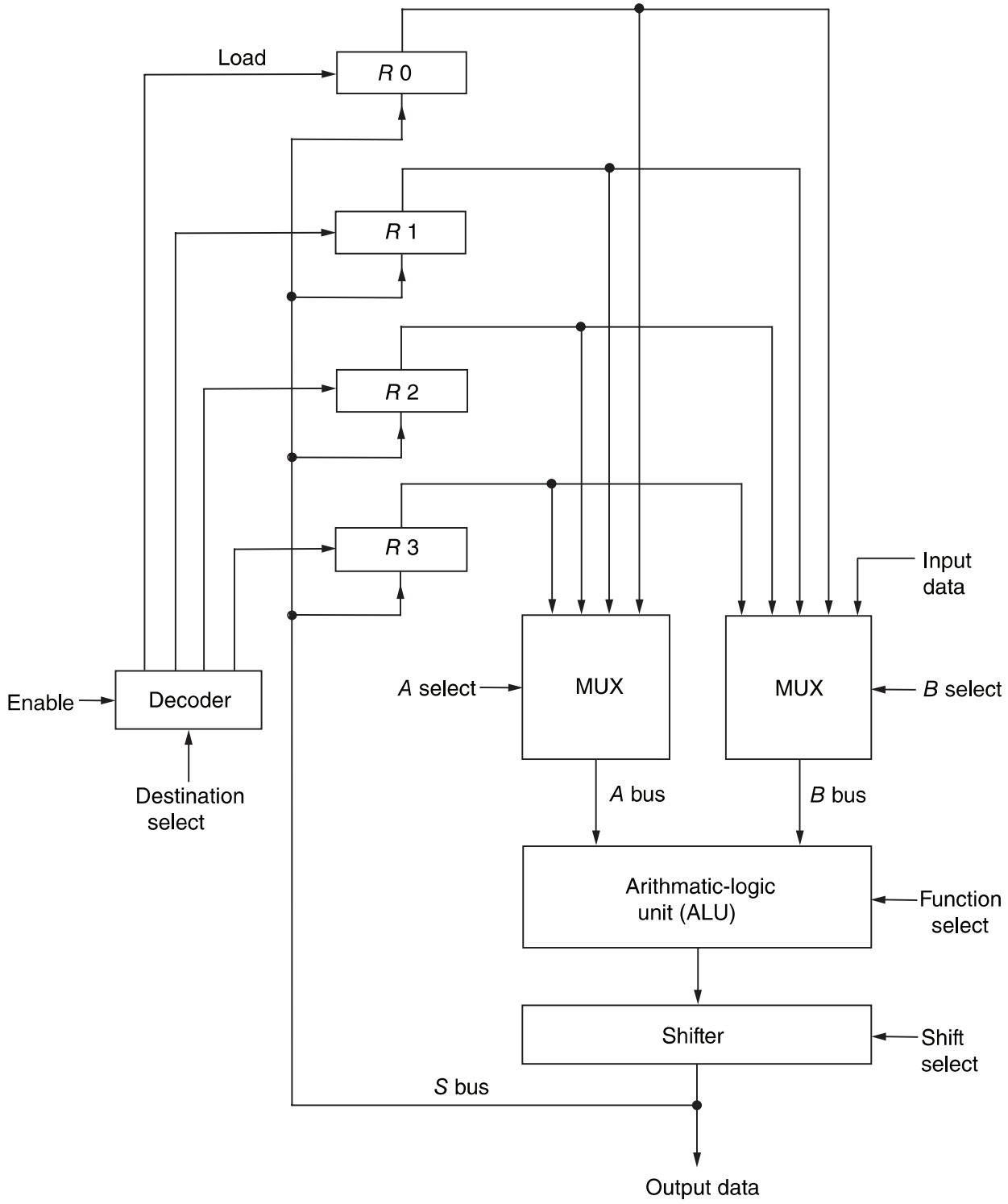


Figure 9-1 Processor registers and ALU connected through common busts

performing various microoperations. A bus organization for four processor registers is shown in Fig. 9-1. Each register is connected to two multiplexers (MUX) to form input buses *A* and *B*. The selection lines of each multiplexer select one register for the particular bus. The *A* and *B* buses are applied to a common arithmetic logic unit. The function selected in the ALU determines the particular operation that is to be performed. The shift microoperations are implemented in

the shifter. The result of the microoperation goes through the output bus *S* into the inputs of all registers. The destination register that receives the information from the output bus is selected by a decoder. When enabled, this decoder activates one of the register load inputs to provide a transfer path between the data on the *S* bus and the inputs of the selected destination register.

The output bus *S* provides the terminals for transferring data to an external destination. One input of multiplexer *A* or *B* can receive data from the outside environment when it is necessary to transfer external data into the processor unit.

The operation of the multiplexers, the buses, and the destination decoder is explained in Section 8-2 in conjunction with Fig. 8-6. The ALU and shifter are discussed later in this chapter.

A processor unit may have more than four registers. The construction of a bus-organized processor with more registers requires larger multiplexers and decoder; otherwise, it is similar to the organization depicted in Fig. 9-1.

The control unit that supervises the processor bus system directs the information flow through the ALU by selecting the various components in the unit. For example, to perform the microoperation:

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to the following selector inputs:

1. MUX *A* selector: to place the contents of *R*2 onto bus *A*.
2. MUX *B* selector: to place the contents of *R*3 onto bus *B*.
3. ALU function selector: to provide the arithmetic operation *A* + *B*.
4. Shift selector: for direct transfer from the output of the ALU onto output bus *S* (no shift).
5. Decoder destination selector: to transfer the contents of bus *S* into *R* 1.

The five control selection variables must be generated simultaneously and must be available during one common clock pulse interval. The binary information from the two source registers propagates through the combinational gates in the multiplexers, the ALU, and the shifter, to the output bus, and into the inputs of the destination register, all during one clock pulse interval. Then, when the next clock pulse arrives, the binary information on the output bus is transferred into *R*1. To achieve a fast response time, the ALU is constructed with carry look-ahead circuits and the shifter is implemented with combinational gates.

When enclosed in an IC package, a processor unit is sometimes called a *register and arithmetic logic unit* or RALU. It is also called by some vendors a *bit-slice microprocessor.* The prefix *micro* refers to the small physical size of the integrated circuit in which (he processor is enclosed. *Bit-slice* refers to the fact that the processor can be expanded to a processor unit with a larger number of bits by using a number of ICs. For example, a 4-bit-stice microprocessor contains registers and ALU for manipulating 4-bit data. Two such ICs can be combined lo construct an 8-bit processor unit. For a 16-bit processor, it is necessary to use four ICs and connect them in cascade. The output carry from one ALU is connected to the input carry of the next higher-order ALU, and the serial output and input lines of the shifters are also connected in cascade. A *bit-slice microprocessor* should be distinguished from another type of IC called a *microprocessor.* The former is a processor unit, whereas a microprocessor refers to an entire computer CPU enclosed in one IC package. Microprocessors and associated equipment are discussed in Chapter 12.

9.2.2 Scratchpad Memory

The registers in a processor unit can be enclosed within a small memory unit. When included in a processor unit, a small memory is sometimes called a *scratchpad* memory. The use of a small memory is a cheaper alternative to connecting processor registers through a bus system. The difference between the two systems is the manner in which information is selected for transfer into the ALU. In a bus system, the information transfer is selected by the multiplexers that form the buses. On the other hand, a single register in a group of registers organized as a small memory must be selected by means of an address to the memory unit. A memory register can function just as any other processor register as long as its only function is to hold binary information to be processed in the ALU.

A scratchpad memory should be distinguished from the main memory of the computer. Contrary to the main memory which stores instructions and data, a small memory in a processor unit is merely an alternative to connecting a number of processor registers through a common transfer path. The information stored in the scratchpad memory would normally come from the main memory by means of instructions in the program.

Consider, for example, a processor unit that employs eight registers of 16 bits each. The registers can be enclosed within a small memory of eight words of 16 bits each, or an 8×16 RAM. The eight memory words can be designated $R0$ through $R7$, corresponding to addresses 0 through 7, and constitute the registers for the processor.

A processor unit that uses a scratchpad memory is shown in Fig. 9-2. A source register is selected from memory and loaded into register A. A second source register is selected from memory and loaded into register B. The selection is done by specifying the corresponding word address and activating the memory-read input. The information in A and B is manipulated in the ALU and shifter. The result of the operation is transferred to a memory register by specifying its word address and activating the memory-write input control. The multiplexer in the input of the memory can select input data from an external source.

Assume that the memory has eight words, so that an address must be specified with three bits. To perform the operation:

$$R1 \leftarrow R2 + R3$$

the control must provide binary selection variables to perform the following sequence of three microoperations:

T_1:	$A \leftarrow$ M [010]	read $R2$ into register A
T_2:	$B \leftarrow$ M [011]	read $R3$ into register B
T_3:	M [001] $\leftarrow A + B$	perform operation in ALU and transfer result to R 1

Control function T_1, must supply an address of 010 to the memory and activate the *read* and *load* A inputs. Control function T_2 must supply an address 011 to the memory and activate the *read* and *load* B inputs. Control function T_3 must supply the function code to the ALU and shifter to perform an *add* operation (with no shift), apply an address 001 to the memory, select the output of the shifter for the MUX, and activate the memory *write* input. The symbol M[*xxx*] designates a memory word (or register) specified by the address given in the binary number *xxx*.

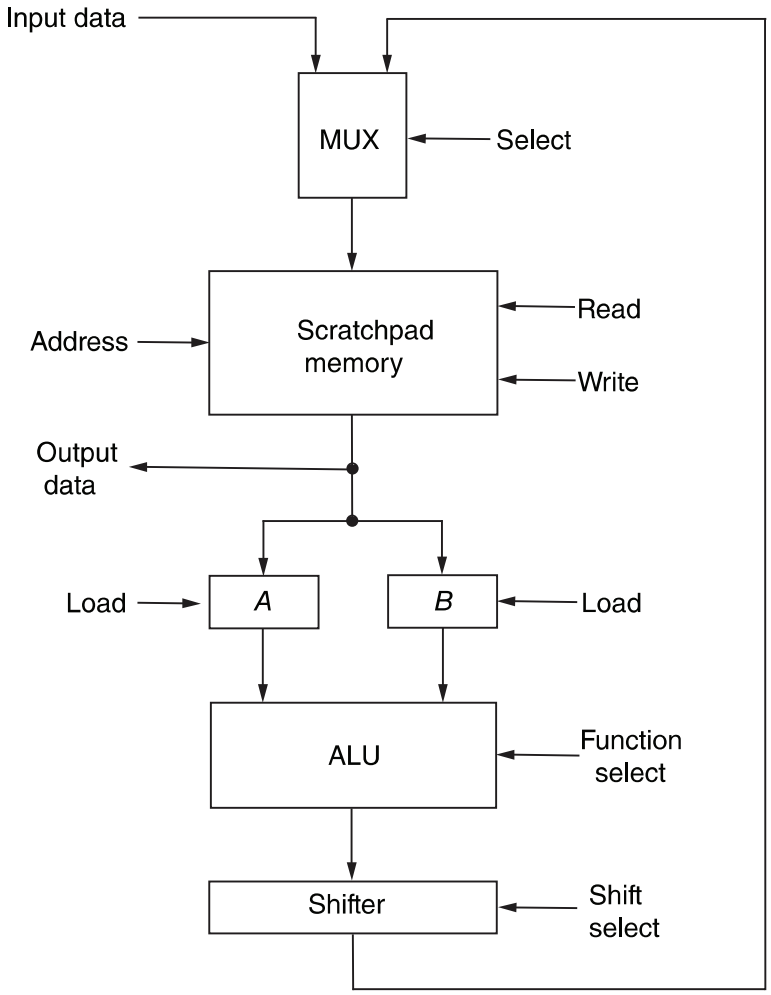


Figure 9-2 Processor unit employing a scratchpad memory

The reason for a sequence of three microoperations, instead of just one as in a bus-organized processor, is due to the limitation of the memory unit. Since the memory unit has only one set of address terminals but two source registers are to be accessed, two accesses to memory are needed to read the source information. The third microoperation is needed to address the destination register. If the destination register is the same as the second source register, the control could activate the read input to extract the second-source information, followed by a write signal to activate the destination transfer, without having to change the address value.

Some processors employ a 2-port memory in order to overcome the delay caused when reading two source registers. A 2-port memory has two separate address lines to select two words of memory simultaneously. In this way, the two source registers can be read at the same time. If the destination register is the same as one of the source registers, then the entire microoperation can be done within one clock pulse period.

The organization of a processor unit with a 2-port scratchpad memory is shown in Fig. 9-3.* The memory has two sets of addresses, one for port *A* and the other for port *B*. Data from any word in memory are read into the *A* register by specifying an *A* address. Likewise, data from any word in memory are read into the *B* register by specifying a *B* address. The same

*This organization is similar to the 4-bit slice microprocessor, type 2901.

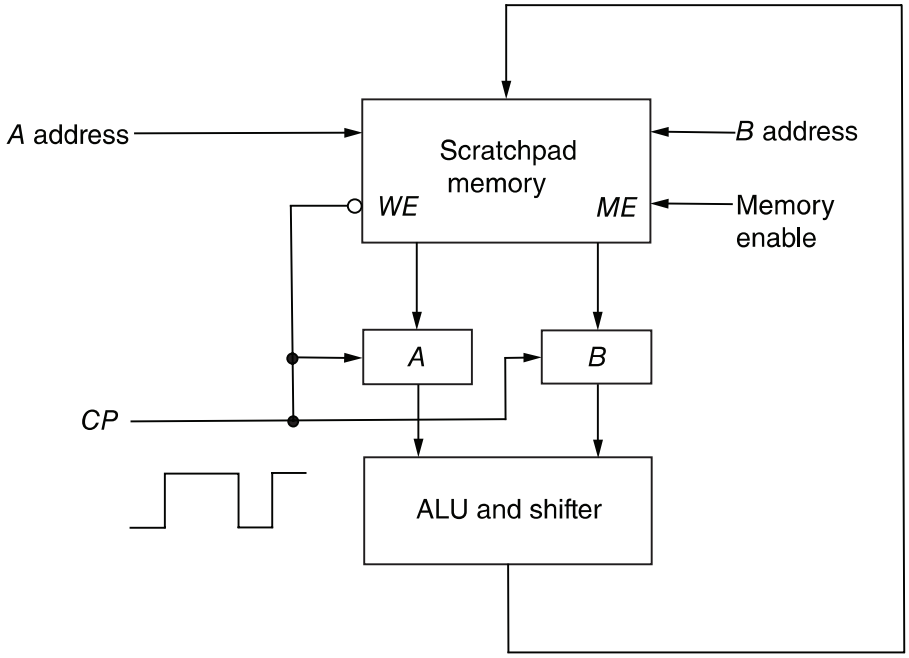


Figure 9-3 Processor unit with a 2-port memory

address can be applied to the *A* address and the *B* address, in which case the identical word will appear in both *A* and *B* registers. When enabled by the memory enable (*ME*) input, new data can be written into the word specified by the *B* address. Thus the *A* and *B* addresses specify two source registers simultaneously, and the *B* address always specifies the destination register. Figure 9-3 does not show a path for external input and output data, but they can be included as in previous organizations.

The *A* and *B* registers are, in effect, latches that accept new information as long as the clock pulse, *CP*, is in the 1-state. When *CP* goes to 0, the latches are disabled, and they hold the information that was stored when *CP* was a 1. This eliminates any possible race conditions that could occur while new information is being written into memory. The clock input controls the memory read and write operations through the write enable (*WE*) input. It also controls the transfers into the *A* and *B* latches. The waveform of one clock pulse interval is shown in the diagram.

When the clock input is 1, the *A* and *B* latches are open and accept the information coming from memory. The *WE* input is also in the 1-state. This disables the write operation and enables the read operation in the memory. Thus, when $CP = 1$, the words selected by the *A* and *B* addresses are read from memory and placed in registers *A* and *S*, respectively. The operation in the ALU is performed with the data stored in *A* and *B*. When the clock input goes to 0, the latches are closed and they retain the last data entered. If the *ME* input is enabled while $WE = 0$, the result of the microoperation is written into the memory word defined by the *B* address. Thus, a microoperation:

$$R1 \leftarrow R1 + R2$$

can be done within one clock pulse period. Memory register *R*1 must be specified with the *B* address, and *R*2 with the *A* address.

9.2.3 Accumulator Register

Some processor units separate one register from all others and call it an *accumulator* register, abbreviated *AC* or *A* register. The name of this register is derived from the arithmetic addition process encountered in digital computers. The process of adding many numbers is carried out by initially storing these numbers in other processor registers or in the memory unit of the computer and clearing the accumulator to 0. The numbers are then added to the accumulator one at a time, in consecutive order. The first number is added to 0, and the sum transferred to the accumulator. The second number is added to the contents of the accumulator, and the newly formed sum replaces its previous value. This process is continued until all numbers are added and the total sum is formed. Thus, the register "accumulates" the sum in a step-by-step manner by performing sequential additions between a new number and the previously accumulated sum.

The accumulator register in a processor unit is a multipurpose register capable of performing not only the add microoperation, but many other microoperations as well. In fact, the gates associated with an accumulator register provide all the digital functions found in an ALU.

Figure 9-4 shows the block diagram of a processor unit that employs an accumulator register. The *A* register is distinguished from all other processor registers. In some cases the entire processor unit is just the accumulator register and its associated ALU. The register itself can function as a shift register to provide the shift microoperations. Input *B* supplies one external source information. This information may come from other processor registers or directly from the main memory of the computer. The *A* register supplies the other source information to the

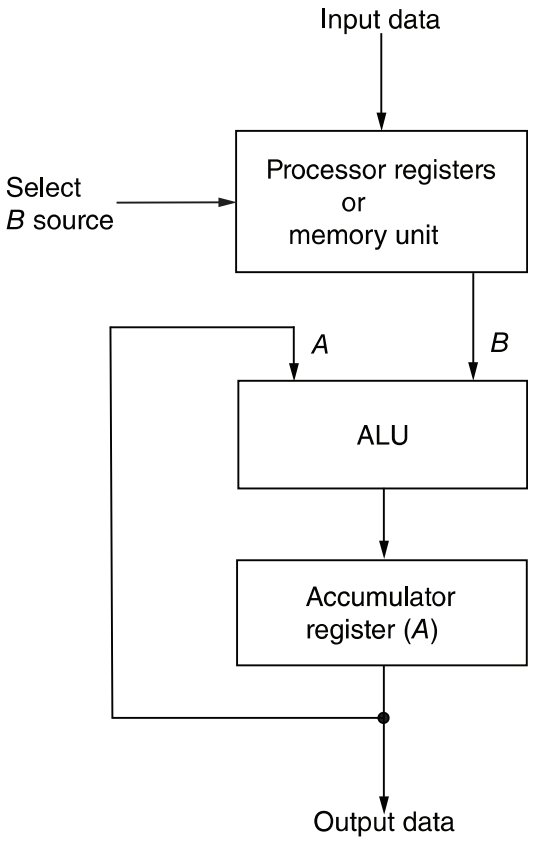


Figure 9-4 Processor with an accumulator register

ALU at input *A*. The result of an operation is transferred back to the *A* register and replaces its previous content. The output from the *A* register may go to an external destination or into the input terminals of other processor registers or memory unit.

To form the sum of two numbers stored in processor registers, it is necessary to add them in the *A* register using the following sequence of microoperations:

T_1:	$A \leftarrow 0$	clear A
T_2:	$A \leftarrow A + R1$	transfer $R1$ to A
T_3:	$A \leftarrow A + R2$	add $R2$ to A

Register *A* is first cleared. The first number in *R*1 is transferred into the *A* register by adding it to the present zero content of *A*. The second number in *R*2 is then added to the present value of *A*. The sum formed in *A* may be used for other computations or may be transferred to a required destination.

9.3 Arithmetic Logic Unit

An arithmetic logic unit (ALU) is a multioperation, combinational-logic digital function. It can perform a set of basic arithmetic operations and a set of logic operations. The ALU has a number of selection lines to select a particular operation in the unit. The selection lines are decoded within the ALU so that k selection variables can specify up to 2^k distinct operations.

Figure 9-5 shows the block diagram of a 4-bit ALU. The four data inputs from *A* are combined with the four inputs from *B* to generate an operation at the *F* outputs. The mode-select input s_2 distinguishes between arithmetic and logic operations. The two function-select inputs s_1 and s_0 specify the particular arithmetic or logic operation to be generated. With three selection variables, it is possible to specify four arithmetic operations (with s_2 in one state) and four logic operations (with s_2 in the other state). The input and output carries have meaning only during an arithmetic operation.

The input carry in the least significant position of an ALU is quite often used as a fourth selection variable that can double the number of arithmetic operations. In this way, it is possible to generate four more operations, for a total of eight arithmetic operations.

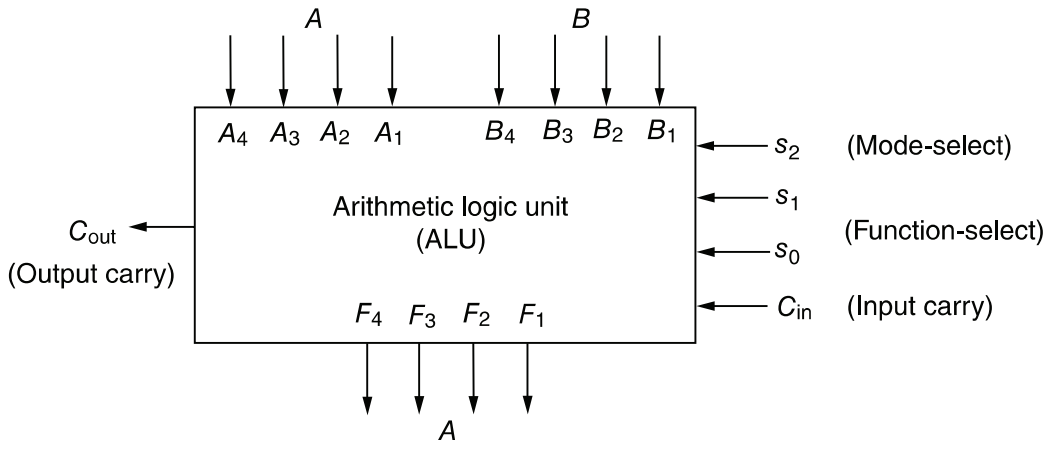


Figure 9-5 Block diagram of a 4-bit ALU

The design of a typical ALU will be carried out in three stages. First, the design of the arithmetic section will be undertaken. Second, the design of the logic section will be considered. Finally, the arithmetic section will be modified so that it can perform both arithmetic and logic operations.

9.4 Design of Arithmetic Circuit

The basic component of the arithmetic section of an ALU is a parallel adder. A parallel adder is constructed with a number of full-adder circuits connected in cascade (see Section 5-2). By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations. Figure 9-6 demonstrates the arithmetic operations obtained when one set of inputs to

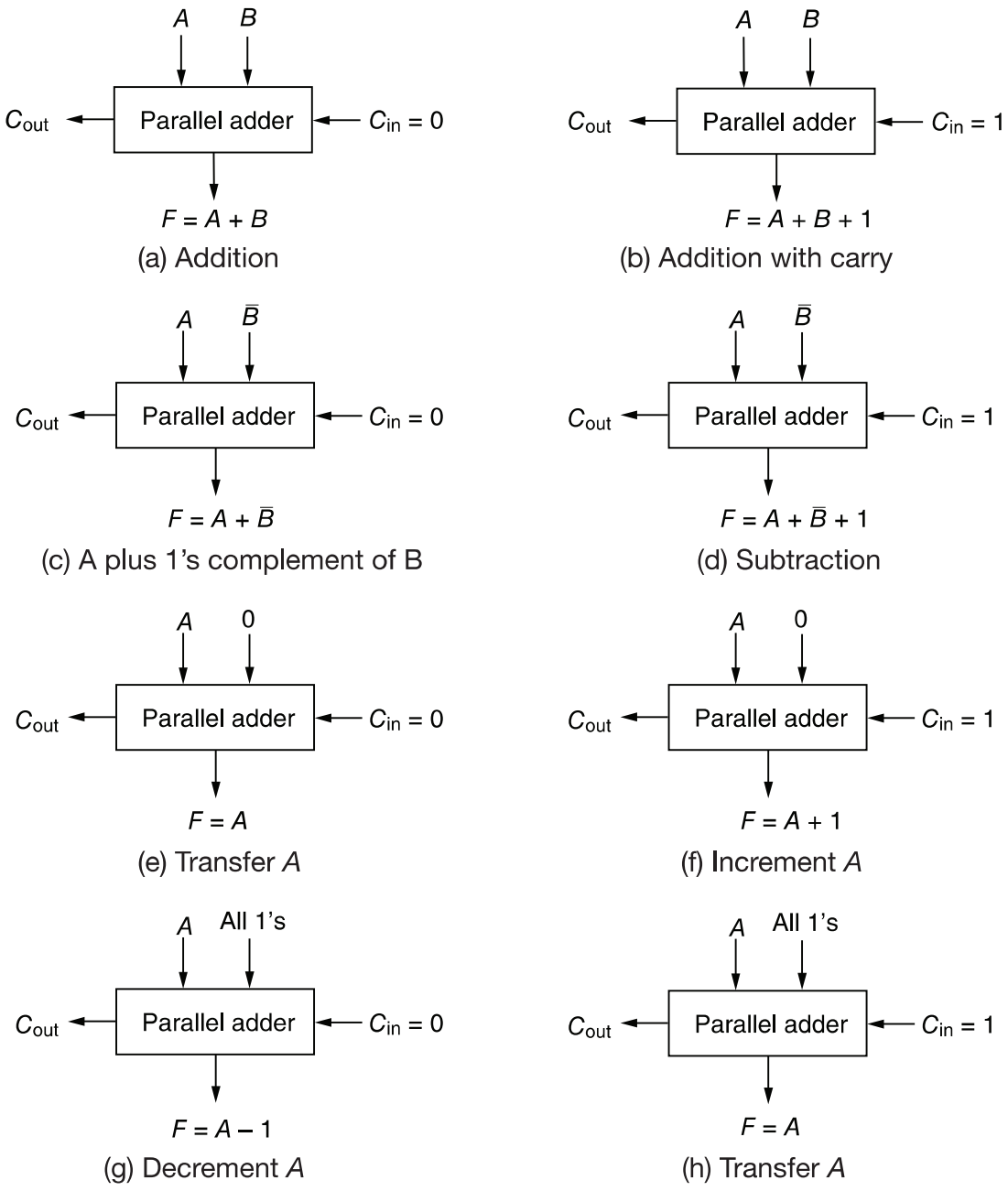


Figure 9-6 Operations obtained by controlling one set of inputs to a parallel adder

a parallel adder is controlled externally. The number of bits in the parallel adder may be of any value. The input carry C_{in} goes to the full-adder circuit in the least significant bit position. The output carry C_{out} comes from the full-adder circuit in the most significant bit position.

The arithmetic addition is achieved when one set of inputs receives a binary number A, the other set of inputs receives a binary number B, and the input carry is maintained at 0. This is shown in Fig. 9-6(a). By making $C_{in} = 1$ as in Fig. 9-6(b), it is possible to add 1 to the sum in F. Now consider the effect of complementing all the bits of input B. With $C_{in} = 0$, the output produces $F = A + \overline{B}$, which is the sum of A plus the 1's complement of B. Adding 1 to this sum by making $C_{in} = 1$, we obtain $F = A + \overline{B} + 1$, which produces the sum of A plus the 2's complement of B. This operation is similar to a subtraction operation if the output carry is discarded. If we force all 0's into the B terminals, we obtain $F = A + 0 = A$, which transfers input A into output F. Adding 1 through C_{in} as in Fig. 9-6(f), we obtain $F = A + 1$, which is the increment operation.

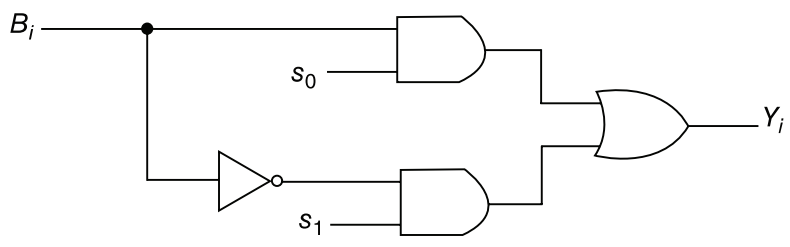
The condition illustrated in Fig. 9-6(g) inserts all 1's into the B terminals. This produces the decrement operation $F = A - 1$. To show that this condition is indeed a decrement operation, consider a parallel adder with n full-adder circuits. When the output carry is 1, it represents the number 2^n because 2^n in binary consists of a 1 followed by n 0's. Subtracting 1 from 2^n, we obtain $2^n - 1$, which in binary is a number of n 1's. Adding $2^n - 1$ to A, we obtain $F = A + 2^n - 1 = 2^n + A - 1$. If the output carry 2^n is removed, we obtain $F = A - 1$.

To demonstrate with a numerical example, let $n = 8$ and $A = 9$. Then:

$A =$	0000	$1001 = (9)_{10}$
$2^n = 1$	0000	$0000 = (256)_{10}$
$2^n - 1 =$	1111	$1111 = (255)_{10}$
$A + 2^n - 1 = 1$	0000	$1000 = (256 + 8)_{10}$

Removing the output carry $2^n = 256$, we obtain $8 = 9 - 1$. Thus, we have decremented A by 1 by adding to it a binary number with all 1's.

The circuit that controls input B to provide the functions illustrated in Fig. 9-6 is called a *true/complement, one/zero* element. This circuit is illustrated in Fig. 9-7. The two selection lines s_1 and s_0 control the input of each B terminal. The diagram shows one typical input designated by B, and an output designated by Y_i. In a typical application, there are n such circuits for $i = 1, 2, \ldots, n$. As shown in the table of Fig. 9-7, when both s_1 and s_0 are equal to 0, the output $Y_i = 0$, regardless of the value of B_i. When $s_1 s_0 = 01$, the top AND gate generates the value of B, while the bottom gate output is 0; so $Y_i = B_i$. With $s_1 s_0 = 10$, the bottom AND gate generates the complement of B_i to give $Y_i = B'_i$. When $s_1 s_0 = 11$, both gates are active and $Y_i = B_i + B'_i = 1$.



s_1	s_0	Y_i
0	0	0
0	1	B_i
1	0	B'_i
1	1	1

Figure 9-7 True/complement, one/zero circuit

Table 9-1 Function table for the arithmetic circuit of Fig. 9-8

Function select			Y equals	Output equals	Function
s_1	s_0	C_{in}			
0	0	0	0	$F = A$	Transfer A
0	0	1	0	$F = A + 1$	Increment A
0	1	0	B	$F = A + B$	Add B to A
0	1	1	B	$F = A + B + 1$	Add B to A plus 1
1	0	0	$\overline{B}$	$F = A + \overline{B}$	Add 1's complement of B to A
1	0	1	$\overline{B}$	$F = A + \overline{B} + 1$	Add 2's complement of B to A
1	1	0	All 1's	$F = A - 1$	Decrement A
1	1	1	All 1's	$F = A$	Transfer A

A 4-bit arithmetic circuit that performs eight arithmetic operations is shown in Fig. 9-8, The four full-adder (FA) circuits constitute the parallel adder. The carry into the first stage is the input carry. The carry out of the fourth stage is the output carry. All other carries are connected internally from one stage to the next. The selection variables are s_1 s_0, and C_{in}. Variables s_1 and s_0 control all of the B inputs to the full-adder circuits as in Fig. 9-7. The A inputs go directly to the other inputs of the full adders.

The arithmetic operations implemented in the arithmetic circuit are listed in Table 9-1. The values of the Y inputs to the full-adder circuits are a function of selection variables s_1 and s_0. Adding the value of Y in each case to the value of A plus the C_{in} value gives the arithmetic operation in each entry. The eight operations listed in the table follow directly from the function diagrams illustrated in Fig. 9-6.

This example demonstrates the feasibility of constructing an arithmetic circuit by means of a parallel adder. The combinational circuit that must be inserted in each stage between the external inputs A_i and B_i and the inputs of the parallel adder X_i and Y_i is a function of the arithmetic operations that are to be implemented. The arithmetic circuit of Fig. 9-8 needs a combinational circuit in each stage specified by the Boolean functions:

$$X_i = A_i$$
$$Y_i = B_i s_0 + B'_i s_i \qquad i = 1, 2, \ldots. n$$

where n is the number of bits in the arithmetic circuit. In each stage i, we use the same common selection variables s_1 and s_0. The combinational circuit will be different if the circuit generates different arithmetic operations.

9.4.1 Effect of Output Carry

The output carry of an arithmetic circuit or ALU has special significance, especially after a subtraction operation. To investigate the effect of the output carry, we expand the arithmetic circuit of Fig. 9-8 to n bits so that $C_{out} = 1$ when the output of the circuit is equal to or greater than 2^n. Table 9-2 lists the conditions for having an output carry in the circuit. The function $F = A$ will

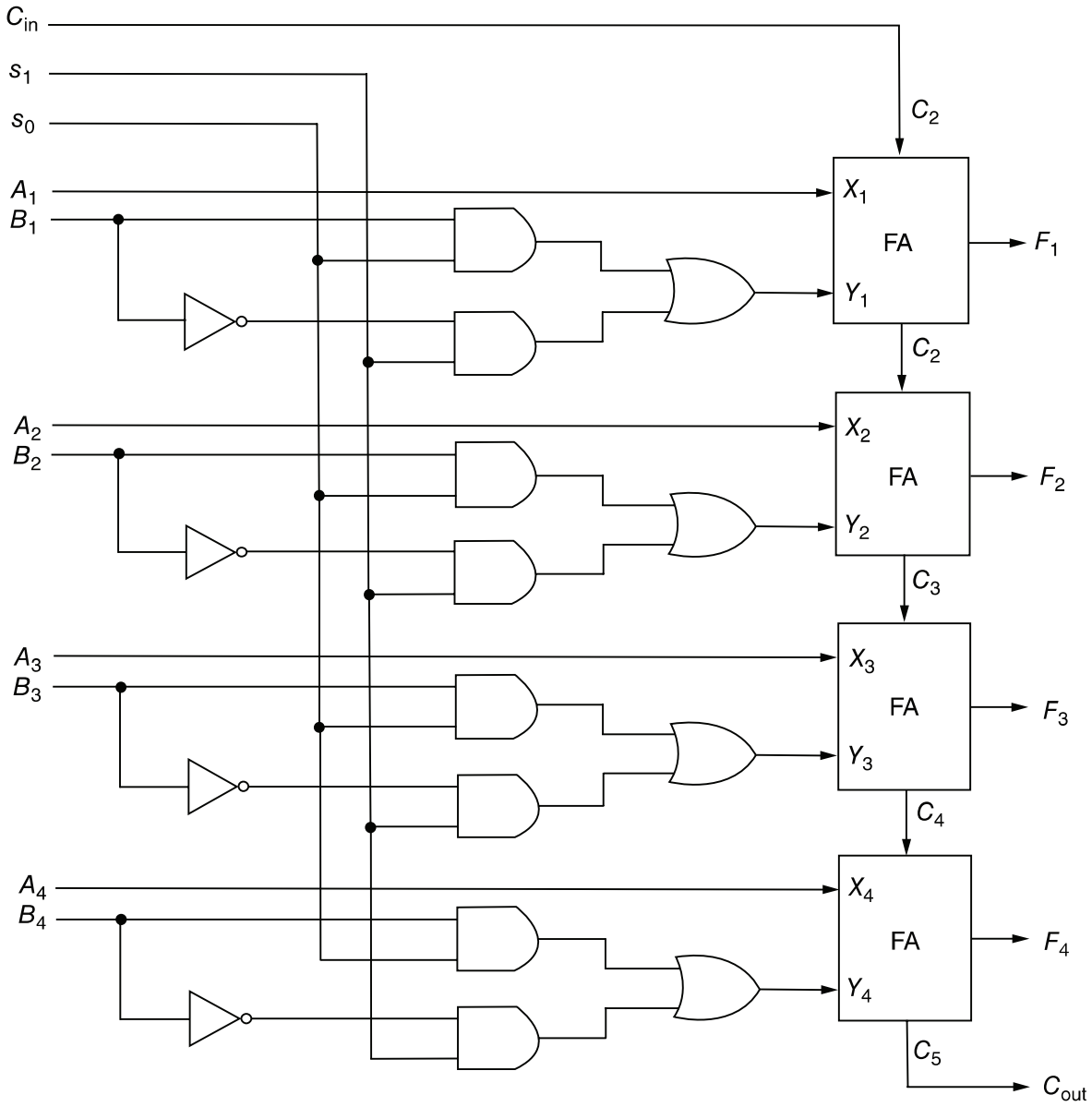


Figure 9-8 Logic diagram of arithmetic circuit

always have the output carry equal to 0. The same applies to the increment operation $F = A + 1$, except when it goes from an all-1's condition to an all-0's condition, at which time it produces an output carry of 1. An output carry of 1 after an addition operation denotes an overflow condition. It indicates that the sum is greater than or equal to 2^n and that the sum consists of $n + 1$ bits.

The operation $F = A + \overline{B}$ adds the 1's complement of B to A. Remember from Section 1-5 that the complement of B can be expressed arithmetically as $2^n - 1 - B$. The arithmetic result in the output will be:

$$F = A + 2^n - 1 - B = 2^n + A - B - 1$$

If $A > B$, then $(A - B) > 0$ and $F > (2^n - 1)$, so that $C_{out} = 1$. Removing the output carry 2^n from this result gives:

$$F = A - B - 1$$

Table 9-2 Effect of output carry in the arithmetic circuit of Fig. 9-8

Function select			Arithmetic function	$C_{out} = 1$ if	Comments
s_1	s_0	C_{in}			
0	0	0	$F = A$		C_{out} is always 0
0	0	1	$F = A + 1$	$A = 2^n - 1$	$C_{out} = 1$ and $F = 0$ if $A = 2^n - 1$
0	1	0	$F = A + B$	$(A + B) \geqslant 2^n$	Overflow occurs if $C_{out} = 1$
0	1	1	$F = A + B + 1$	$(A + B) \geqslant (2^n - 1)$	Overflow occurs if $C_{out} = 1$
1	0	0	$F = A - B - 1$	$A > B$	If $C_{out} = 0$, then $A < B$ and F = 1's complement of $(B - A)$
1	0	1	$F = A - B$	$A \geqslant B$	If $C_{out} = 0$, then $A < B$ and F = 2's complement of $(B - A)$
1	1	0	$F = A - 1$	$A \neq 0$	C_{out} = l, except when $A = 0$
1	1	1	$F = A$		C_{out} is always 1

which is a subtraction with borrow. Note that if $A \leqslant B$, then $(A - B) \leqslant 0$ and $F \leqslant (2^n - 1)$, so that $C_{out} = 0$. For this condition it is more convenient to express the arithmetic result as:

$$F = (2^n - 1) - (\mathrm{B} - A)$$

which is the l's complement of $B - A$.

The condition for output carry when $F = A + \overline{B} + 1$ can be derived in a similar manner. $B + 1$ is the symbol for the 2's complement of B. Arithmetically, this is an operation that produces a number equal to $2^n - B$. The result of the operation can be expressed as:

$$F = A + 2^n - B = 2^n + A - B$$

If $A \geqslant B$, then $(A - B) \geqslant 0$ and $F \geqslant 2^n$, so that $C_{out} = 1$. Removing the output carry 2^n, we obtain:

$$F = A - B$$

which is a subtraction operation. If, however, $A < B$, then $(A - B) < 0$ and $F < 2^n$, so that $C_{out} = 0$. The arithmetic result for this condition can be expressed as:

$$F = 2^n - (B - A)$$

which is the 2's complement of $\mathrm{B} - A$. Thus, the output of the arithmetic subtraction is correct as long as $A \geqslant B$. The output should be $B - A$ if $B > A$, but the circuit generates the 2's complement of this number.

The decrement operation is obtained from $F = A + (2^n - 1) = 2^n + A - 1$. The output carry is always 1 except when $A = 0$. Subtracting 1 from 0 gives -1, and -1 in 2's complement is $2^n - 1$, which is a number with all l's. The last entry in Table 9-2 generates $F = (2^n - 1) + A + 1 = 2^n + A$. This operation transfers A into F and gives an output carry of 1.

9.4.2 Design of Other Arithmetic Circuits

The design of any arithmetic circuit that generates a set of basic operations can be undertaken by following the procedure outlined in the previous example. Assuming that all operations in the set can be generated through a parallel adder, we start by obtaining a function diagram as in Fig. 9-6. From the function diagram, we obtain a function table that relates the inputs of the full-adder circuit to the external inputs. From the function table, we obtain the combinational gates that must be added to each full-adder stage. This procedure is demonstrated in the following example.

EXAMPLE 9-1: Design an adder/subtractor circuit with one selection variable s and two inputs A and B. When $s = 0$ the circuit performs $A + B$. When $s = 1$ the circuit performs $A - B$ by taking the 2's complement of B.

The derivation of the arithmetic circuit is illustrated in Fig. 9-9. The function diagram is shown in Fig. 9-9(a). For the addition part, we need $C_{in} = 0$. For the subtraction part, we need the complement of B and $C_{in} = 1$. The function table is listed in Fig. 9-9(b). When $s = 0$, X_i and Y_i of each full adder must be equal to the external inputs A_i and B_i, respectively. When $s = 1$, we must have $X_i = A$, and $Y_i = B'_i$. The input carry must be equal to the value of s. The diagram in (b) shows the position of the combinational circuit in one typical stage of the arithmetic circuit. The truth table in (c) is obtained by listing the eight values of the binary input variables. Output X_i is made to be equal to input A_i in all eight entries. Output Y_i is equal to B_i for the four entries when $s = 0$. It is equal to the complement of B_i for the last four entries where $s = 1$. The simplified output functions for the combinational circuit are:

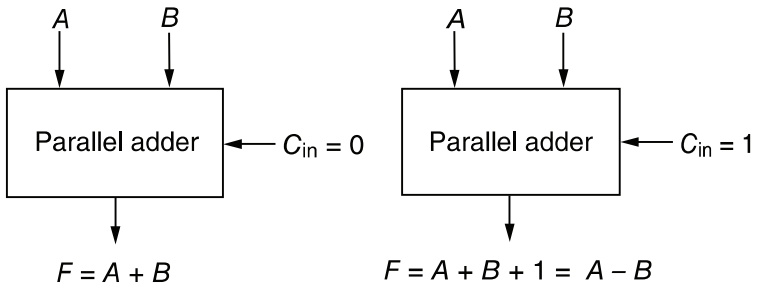
$$X_i = A_i$$
$$Y_i = B_i \oplus s$$

The diagram of the 4-bit adder/sub tractor circuit is shown in Fig. 9-10. Each input B_i requires an exclusive-OR gate. The selection variable s goes to one input of each gate and also to the input carry of the parallel adder. The 4-bit adder/subtractor can be constructed with two ICs. One IC is the 4-bit parallel adder and the other is a quadruple exclusive-OR gates.

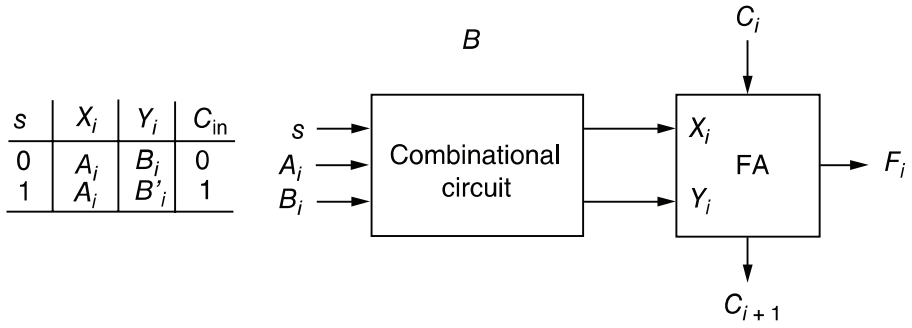
9.5 Design of Logic Circuit

The logic microoperations manipulate the bits of the operands separately and treat each bit as a binary variable. Table 2-6 listed 16 logic operations that can be performed with two binary variables. The 16 logic operations can be generated in one circuit and selected by means of four selection lines. Since all logic operations can be obtained by means of AND, OR, and NOT (complement) operations, it may be more convenient to employ a logic circuit with just these operations. For three operations, we need two selection variables. But two selection Lines can select among four logic operations, so we choose also the exclusive-OR (XOR) function for the logic circuit to be designed in this and the next section.

The simplest and most straightforward way to design a logic circuit is shown in Fig. 9-11. The diagram shows one typical stage designated by subscript i. The circuit must be repeated n times for an n-bit logic circuit. The Four gates generate the four logic operations OR. XOR,



(a) Function specification



(b) Specifying combinational circuit

s	A_i	B_i	X_i	Y_i
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	0
1	1	0	1	1
1	1	1	1	0

$X_i = A_i$

$Y_i = B_i \oplus s$

$C_{in} = s$

(c) Truth table and simplified equations

Figure 9.9 Derivation of an adder/subtractor circuit

AND, and NOT. The two selection variables in the multiplexer select one of the gates for the output. The function table lists the output logic generated as a function of the two selection variables.

The logic circuit can be combined with the arithmetic circuit to produce one arithmetic logic unit. Selection variables s_1 and s_0 can be made common to both sections provided we use a third selection variable, s_2, to differentiate between the two. This configuration is illustrated in Fig. 9-12. The outputs of the logic and arithmetic circuits in each stage go through a multiplexer with selection variable s_2. When $s_2 = 0$, the arithmetic output is selected, but when $s_2 = 1$, the logic output is selected. Although the two circuits can be combined in this manner, this is not the best way to design an ALU.

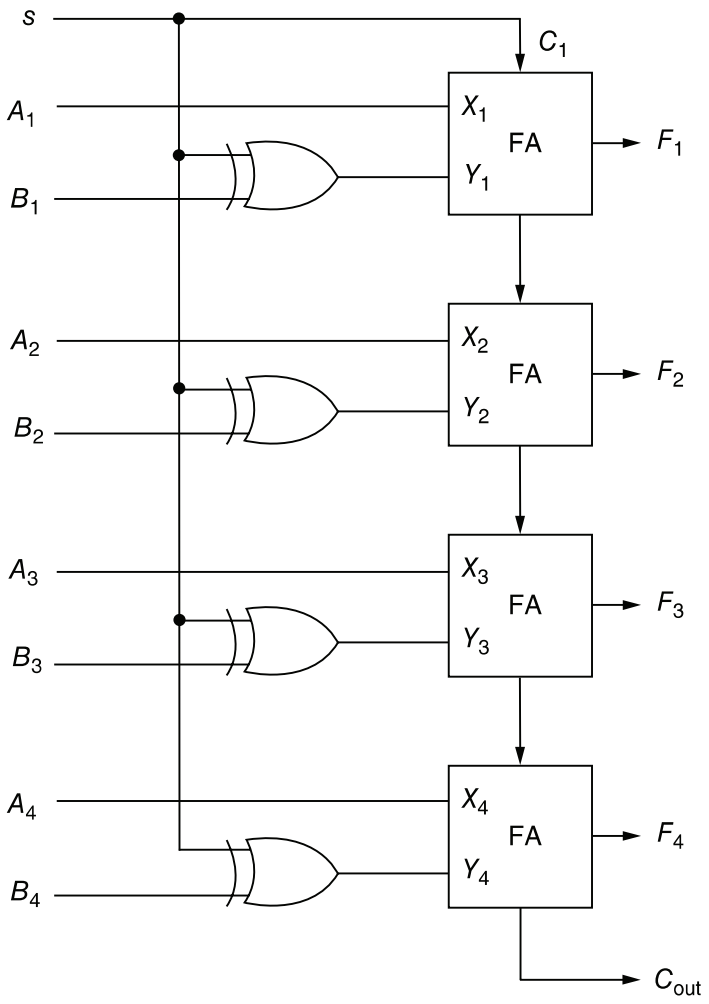
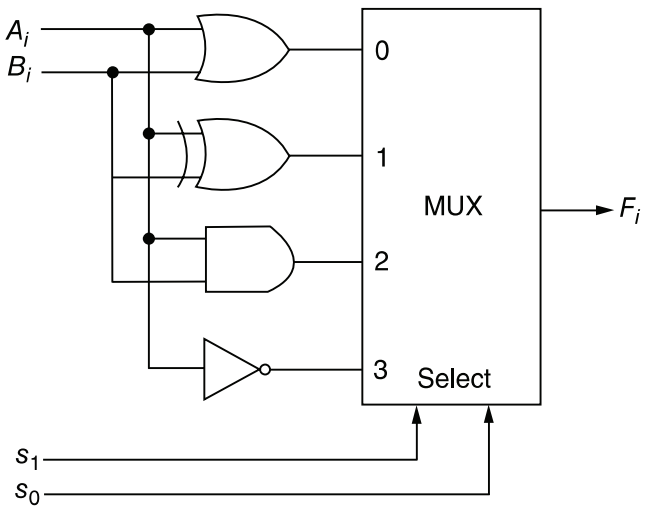


Figure 9-10 4-bit adder/subtractor circuit



(a) Logic diagram

s_1	s_0	Output	Operation
0	0	$F_i = A_i + B_i$	OR
0	1	$F_i = A_i \oplus B_i$	XOR
1	0	$F_i = A_i B_i$	AND
1	1	$F_i = A'_i$	NOT

(b) Function table

Figure 9-11 One stage of logic circuit

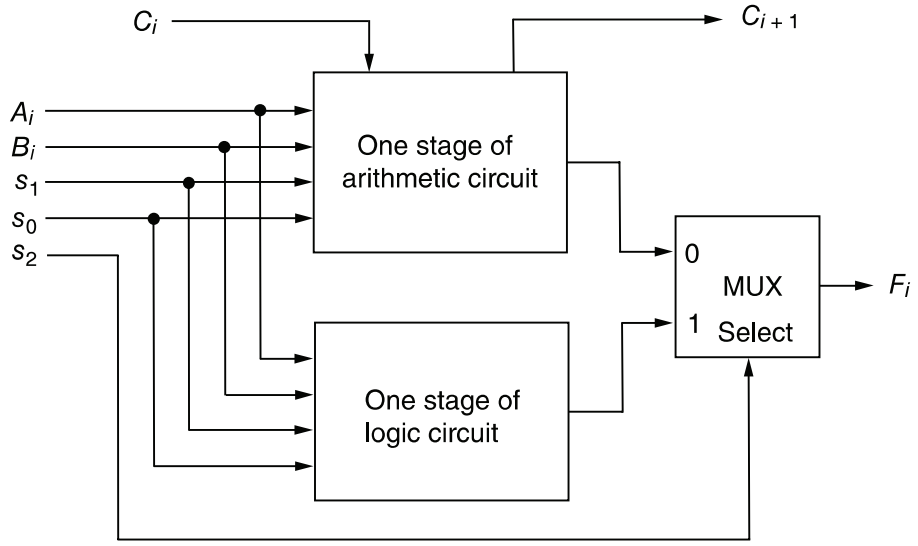


Figure 9-12 Combining logic and arithmetic circuits

A more efficient ALU can be obtained if we investigate the possibility of generating logic operations in an already available arithmetic circuit. This can be done by inhibiting all input carries into the full-adder circuits of the parallel adder. Consider the Boolean function that generates the output sum in a full-adder circuit:

$$F_i = X_i \oplus Y_i \oplus C_i$$

The input carry C_i in each stage can be made to be equal to 0 when a selection variable s_2 is equal to 1. The result would be:

$$F_i = X_i \oplus Y_i$$

This expression is valid because of the property of the exclusive-OR operation $x \oplus 0 = x$. Thus, with the input carry to *each* stage equal to 0, the full-adder circuits generate the exclusive-OR operation.

Now consider the arithmetic circuit of Fig. 9-8. The value of Y_i can be selected by means of the two selection variables to be equal to either 0, B_i, B'_i or 1.

The value of X_i is always equal to input A_i. Table 9-3 shows the four logic operations obtained when a third selection variable $s_2 = 1$. This selection variable forces C_i to be equal to

Table 9-3 Logic operations in one stage of arithmetic circuit

s_2	s_1	s_0	X_i	Y_i	C_i	$F_i = X_i \oplus Y_i$	Operation	Required operation
1	0	0	A_i	0	0	$F_i = A_i$	Transfer A	OR
1	0	1	A_i	B_i	0	$F_i = A_i \oplus B_i$	XOR	XOR
1	1	0	A_i	B'_i	0	$F_i = A_i \odot B_i$	Equivalence	AND
1	1	1	A_i	1	0	$F_i = A'_i$	NOT	NOT

0 while s_1 and s_0 choose a particular value for Y_i. The four logic operations obtained by this configuration are transfer, exclusive-OR, equivalence, and complement. The third entry is the equivalence operation because:

$$A_i \oplus B'_i = A_i B_i + A'_i B'_i = A_i \odot B_i$$

The last entry in the table is the NOT or complement operation because:

$$A_i \oplus 1 = A'_i$$

The table has one more column which lists the four logic operations we want to include in the ALU. Two of these operations, XOR and NOT, are already available. The question that must be answered is whether it is possible to modify the arithmetic circuit further so that it will generate the logic functions OR and AND instead of the transfer and equivalence functions. This problem is investigated in the next section.

9.6 Design of Arithmetic Logic Unit

En this section, we design an ALU with eight arithmetic operations and four logic operations. Three selection variables s_2, s_1 and s_0 select eight different operations, and the input carry C_{in} is used to select four additional arithmetic operations. With $s_2 = 0$, selection variables s_1 and s_0 together with C_{in} will select the eight arithmetic operations listed in Table 9-1. With $s_2 = 1$, variables s_1 and s_0 will select the four logic operations OR, XOR, AND, and NOT.

The design of an ALU is a combinational-logic problem. Because the unit has a regular pattern, it can be broken into identical stages connected in cascade through the carries. We can design one stage of the ALU and then duplicate it for the number of stages required. There are six inputs to each stage: A_i, B_i, C_i, s_2, s_1, and s_0. There are two outputs in each stage: output F_i and the carry out C_{i+1}. One can formulate a truth table with 64 entries and simplify the two output functions.

Here we choose to employ an alternate procedure that uses the availability of a parallel adder.

The steps involved in the design of an ALU are as follows:

1. Design the arithmetic section independent of the logic section.
2. Determine the logic operations obtained from the arithmetic circuit in step 1, assuming that the input carries to all stages are 0.
3. Modify the arithmetic circuit to obtain the required logic operations.

The third step in the design is not a straight forward procedure and requires a certain amount of ingenuity on the part of the designer. There is no guarantee that a solution can be found or that the solution uses the minimum number of gates. The example presented here demonstrates the type of logical thinking sometimes required in the design of digital systems.

It must be realized that various ALUs are available in EC packages. In a practical situation, all that one must do is search for a suitable ALU or processor unit among the ICs that are available commercially. Yet, the internal logic of the IC selected must have been designed by a person familiar with logic design techniques.

The solution to the first design step is shown in Fig. 9-8. The solution to the second design step is presented in Table 9-3. The solution of the third step is carried out below.

From Table 9-3, we see that when $s_2 = 1$, the input carry C_i in each stage must be 0. With $s_1 s_0 = 00$, each stage as it stands generates the function $F_i = A_i$. To change the output to an OR operation, we must change the input to each full-adder circuit from A_i, to $A_i + B_i$. This can be accomplished by ORing B_i and A_i when $s_2 s_1 s_0 = 100$.

The other selection variables that give an undesirable output occur when $s_2 s_1 s_0 = 110$. The unit as it stands generates an output $F_i = A_i \odot B_i$ but we want to generate the AND operation $F_i = A_i B_i$. Let us investigate the possibility of ORing each input A_i with some Boolean function K_i. The function so obtained is then used for X_i when $s_2 s_1 s_0 = 110$:

$$F_i = X_i \oplus Y_i = (A_i \oplus K_i) \oplus B'_i = A_i B_i + K_i B_i + A'_i K'_i B'_i$$

Careful inspection of the result reveals that if the variable $K_i = B'_i$, we obtain an output:

$$F_i = A_i B_i + B'_i B_i + A_i B_i B'_i = A_i B_i$$

Two terms are equal to 0 because $B_i B'_i = 0$. The result obtained is the AND operation as required. The conclusion is that, if A_i is ORed with B'_i when $s_2 s_1 s_0 = 110$, the output will generate the AND operation.

The final ALU is shown in Fig. 9-13. Only the first two stages are drawn, but the diagram can be easily extended to more stages. The inputs to each full-adder circuit are specified by the Boolean functions:

$$X_i = A_i + s_2 s'_1 s'_0 B_i + s_2 s_1 s'_0 B'_i$$
$$Y_i = s_0 B_i + s_1 B'_i$$
$$Z_i = s_2 C'_i$$

When $s_2 = 0$, the three functions reduce to;

$$X_i = A_i$$
$$Y_i = s_0 B_i + s_1 B'_i$$
$$Z_i = C'_i$$

which are the functions for the arithmetic circuit of Fig. 9-8. The logic operations are generated when $s_2 = 1$. For $s_2 s_1 s_0 = 101$ or 111, the functions reduce to:

$$X_i = A_i$$
$$Y_i = s_0 B_i + s_1 B'_i$$
$$C_i = 0$$

Output F_i is then equal to $X_i \oplus Y_i$ and produces the exclusive-OR and complement operations as specified in Table 9-3. When $s_2 s_1 s_0 = 100$, each A_i is ORed with B_i to provide the OR operation as discussed above. When $s_2 s_1 s_0 = 110$, each A_i is ORed with B'_i to provide the AND operation as explained previously.

The 12 operations generated in the ALU are summarized in Table 9-4. The particular function is selected through s_2, s_1, s_0, and C_{in}. The arithmetic operations are identical to the ones listed for the arithmetic circuit. The value of C_{in} for the four logic functions has no effect on the operation of the unit and those entries are marked with don't-care X's.

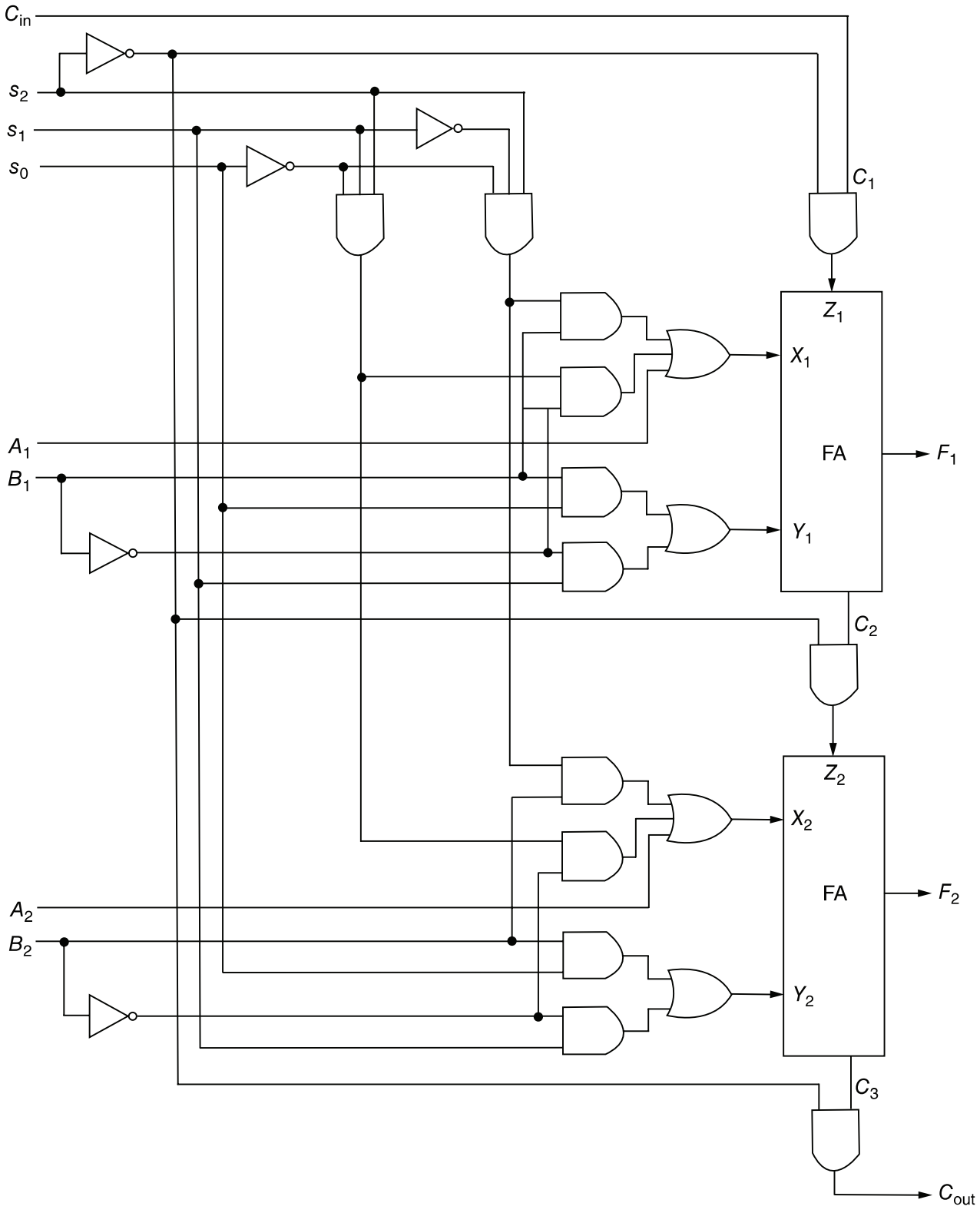


Figure 9-13 Logic diagram of arithmetic logic unit (ALU)

Table 9-4 Function table for the ALU of Fig. 9-13

Selection				Output	Function
s_2	s_1	s_0	C_{in}		
0	0	0	0	$F = A$	Transfer A
0	0	0	1	$F = A + 1$	Increment A
0	0	1	0	$F = A + B$	Addition
0	0	1	1	$F = A + B + 1$	Add with carry
0	1	0	0	$F = A - B - 1$	Subtract with borrow
0	1	0	1	$F = A - B$	Subtraction
0	1	1	0	$F = A - 1$	Decrement A
0	1	1	1	$F = A$	Transfer A
1	0	0	X	$F = A \vee B$	OR
1	0	1	X	$F = A \oplus B$	XOR
1	1	0	X	$F = A \wedge B$	AND
1	1	1	X	$F = \overline{A}$	Complement A

9.7 Status Register

The relative magnitudes of two numbers may be determined by subtracting one number from the other and then checking certain bit conditions in the resultant difference. If the two numbers are unsigned, the bit conditions of interest are the output carry and a possible zero result. If the two numbers include a sign bit in the highest-order position, the bit conditions of interest are the sign of the result, a zero indication, and an overflow condition. It is sometimes convenient to supplement the ALU with a status register where these status-bit conditions are stored for further analysis. Status-bit conditions are sometimes called *condition-code* bits or *flag* bits.

Figure 9-14 shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. Bit C is set if the output carry of the ALU is 1. It is cleared if the output carry is 0.
2. Bit S is set if the highest-order bit of the result in the output of the ALU (the sign bit) is 1. It is cleared if the highest-order bit is 0.
3. Bit Z is set if the output of the ALU contains all 0's, and cleared otherwise. $Z = 1$ if the result is zero, and $Z = 0$ if the result is nonzero.
4. Bit V is set if the exclusive-OR of carries C_8 and C_9 is 1, and cleared otherwise. This is the condition for overflow when the numbers are in sign-2's-complement representation (see Section 8-6). For the 8-bit ALU, V is set if the result is greater than 127 or less than -128.

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that

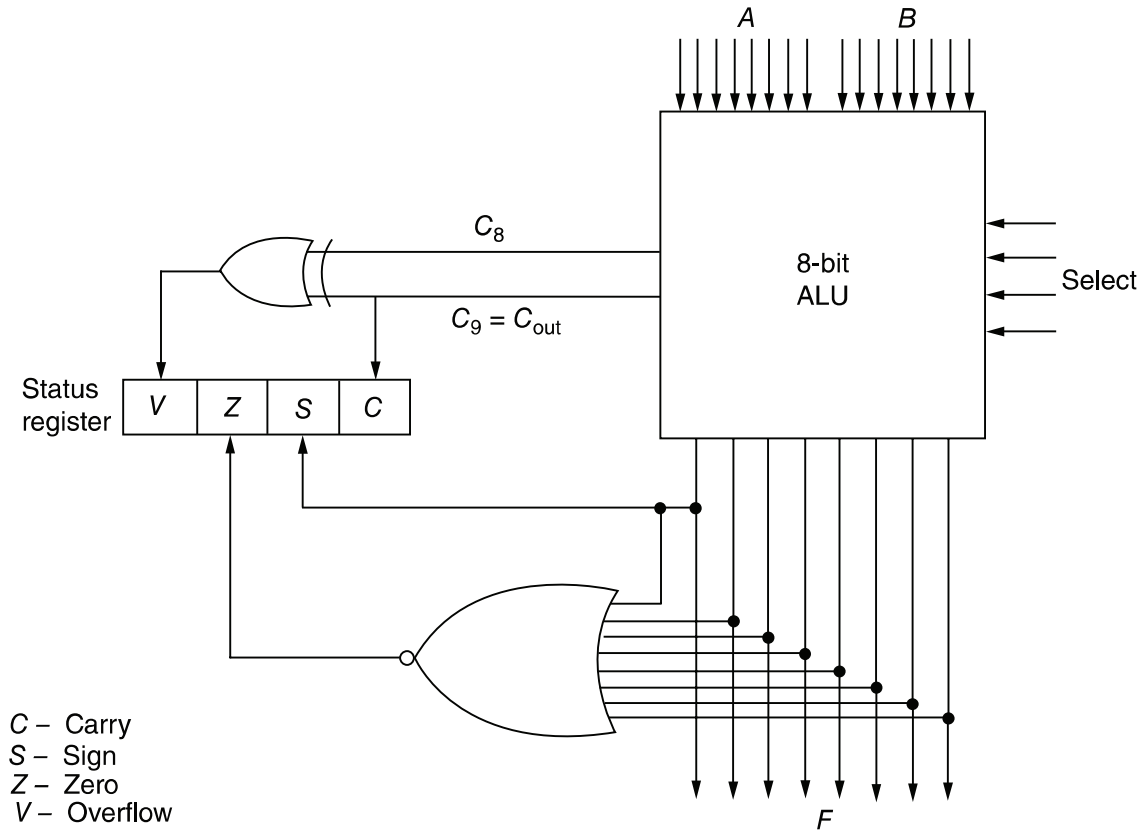


Figure 9-14 Setting bits in a status register

$A = B$. This is so because $x \oplus x = 0$, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit. A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let $A = 101\ x\ 1100$, where x is the bit to be checked. The AND operation of A with $B = 00010000$ produces a result $000\ x\ 0000$. If $x = 0$, the Z status bit is set, but if $x = 1$, the Z bit is cleared since the result is not zero.

The *compare* operation is a subtraction of B from A, except that the result of the operation is not transferred into a destination register, but the status bits are affected. The status register then provides the information about the relative magnitudes of A and B. The status bits to consider depend on whether we take the two numbers to be unsigned or signed and in 2's-complement representation.

Consider the operation $A - B$ done with two *unsigned* binary numbers. The relative magnitudes of A and B can be determined from the values transferred to the C and Z status bits. If $Z = 1$, then we know that $A = B$, since $A - B = 0$. If $Z = 0$, then we know that $A \neq B$. From Table 9-2, we have that $C = 1$ if $A > B$ and $C = 0$ if $A < B$. These conditions are listed in Table 9-5. The table lists two other conditions. For A to be greater than but not equal to B ($A > B$), we must have $C = 1$ and $Z = 0$. Since C is set when the result is 0, we must check Z to ensure that the result is not 0. For A to be less than or equal to B($A \leq B$), the C bit must be 0 (for $A < B$) or the Z bit must be 1 (for $A = B$). Table 9-5 also lists the Boolean functions that must be satisfied for each of the six relationships.

Some computers consider the C bit to be a borrow bit after a subtraction operation $A - B$. An end borrow does not occur if $A \geq B$, but an extra bit must be borrowed when $A < B$. The con-

Table 9-5 Status bits after the subtraction of unsigned numbers ($A - B$)

Relation	Condition of status bits	Boolean function
A > B	$C = 1$ and $Z = 0$	CZ'
$A \geq B$	$C = 1$	C
A <B	$C = 0$	C'
$A \leq B$	$C = 0$ and $Z = 1$	$C' + Z$
$A = B$	$Z = 1$	Z
$A \neq B$	$Z = 0$	Z'

dition for a borrow is the complement of the output carry obtained when the subtraction is done by taking the 2's complement of B. For this reason, a processor that considers the C bit to be a borrow after a subtraction will complement the C bit after a subtraction or compare operation and denote this bit as a borrow.

Now consider the operation $A - B$ done with two *signed* binary numbers when negative numbers are in 2's-complement form. The relative magnitudes of A and B can be determined from the values transferred to the Z, S, and V status bits. If $Z = 1$, then we know that $A = B$; when $Z = 0$, we have that $A' \neq B$. If $S = 0$, the sign of the result is positive, so A must be greater than B. This is true if there was no overflow and $V = 0$. If the result overflows, we obtain an erroneous result. It was shown in Section 8-5 that an overflow condition changes the sign of the result. Therefore, if $S = 1$ and $V = 1$, it indicates that the result should have been positive and therefore A must be greater than B.

Table 9-6 lists the six possible relationships that can exist between A and B and the corresponding values of Z, S, and V in each case. For $A - B$ to be greater than but not equal to zero ($A > B$), the result must be positive and nonzero. Since a zero result gives a positive sign, we must ensure that the Z bit is 0 to exclude the possibility of $A = B$. For $A \geq B$. it is sufficient to check for a positive sign when no overflow occurs or a negative sign when an overflow occurs. For $A < B$, the result must be negative. If the result is negative or zero, we have that $A \leq B$. The Boolean functions listed in the table express the status-bit conditions in algebraic form.

Table 9-6 Status bits after the subtraction of sign-2's complement numbers ($A - B$)

Relation	Condition of status bits	Boolean function
$A > B$	$Z = 0$ and ($S = 0$, $V = 0$ or $S = 1$, $V = 1$)	$Z'(S \odot V)$
$A \geqslant B$	$S = 0$, $V = 0$ or $S = 1$, $V = 1$	$S \odot V$
$A < B$	$S = 1$, $V = 0$ or $S = 0$, $V = 1$	$S \oplus V$
$A \leqslant B$	$S = 1$, $V = 0$ or $S = 0$, $V = 1$ or $Z = 1$	$(S \oplus V) + Z$
$A = B$	$Z = 1$	Z
$A \neq B$	$Z = 0$	Z'

9.8 Design of Shifter

The shift unit attached to a processor transfers the output of the ALU onto the output bus. The shifter may transfer the information directly without a shift, or it may shift the information to the right or left. Provision is sometimes made for no transfer from the ALU to the output bus. The shifter provides the shift microoperations commonly not available in an ALU.

An obvious circuit for a shifter is a bidirectional shift-register with parallel load. The information from the ALU can be transferred to the register in parallel and then shifted to the right or left. In this configuration, a clock pulse is needed for the transfer to the shift register, and another pulse is needed for the shift. These two pulses are in addition to the pulse required to transfer the information from the shift register to a destination register.

The transfer from a source register to a destination register can be done with one clock pulse if the shifter is implemented with a combinational circuit. In a combinational-logic shifter, the signals from the ALU to the output bus propagate through gates without the need for a clock pulse. Hence, the only clock pulse needed in the processor system is for loading the data from the output bus into the destination register.

A combinational-logic shifter can be constructed with multiplexers as shown in Fig. 9-15. The two selection variables, H_1 and H_0, applied to all four multiplexers select the type of operation in the shifter. With $H_1H_0 = 00$, no shift is executed and the signals from F go directly to the S lines. The next two selection variables cause a shift-right operation and a shift-left operation. When $H_1H_0 = 11$, the multiplexers select the inputs attached to 0 and as a consequence the S outputs are also equal to 0, blocking the transfer of information from the ALU to the output bus. Table 9-7 summarizes the operation of the shifter.

The diagram of Fig. 9-15 shows only four stages of the shifter. The shifter, of course, must consist of n stages in a system with n parallel lines. Inputs I_R and I_L serve as serial inputs for the

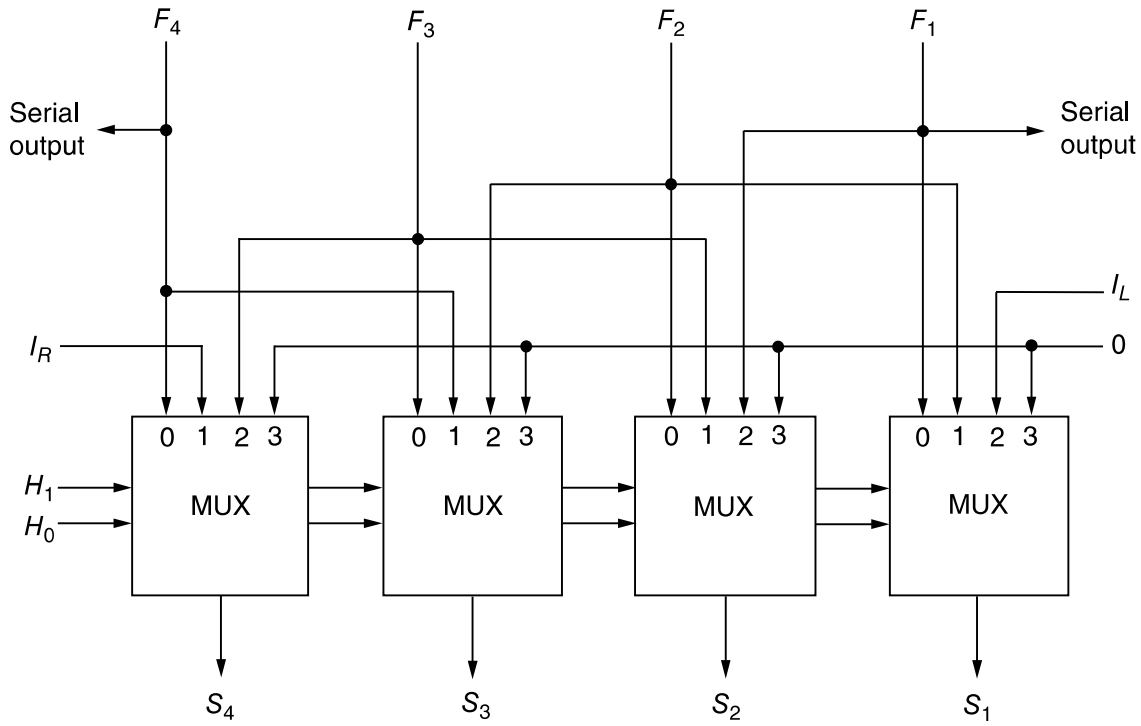


Figure 9-15 4-bit combinational-logic shifter

Table 9-7 Function table for shifter

H_1	H_0	Operation	Function
0	0	$S \leftarrow F$	Transfer F to S (no shift)
0	1	$S \leftarrow$ shr F	Shift-right F into S
I	0	$S \leftarrow$ shl F	Shift-left F into S
1	I	$S \leftarrow 0$	Transfer 0's into S

last and first stages during a shift-right or shift-left, respectively. Another selection variable may be employed to specify what goes into I_R or I_L during the shift. For example, a third selection variable, H_2, when in one state can select a 0 for the serial input during the shift. When H_2 is in the other state, the information can be circulated around together with the value of the carry status bit. In this way, a carry produced during an addition operation can be shifted to the right and into the most significant bit position of a register.

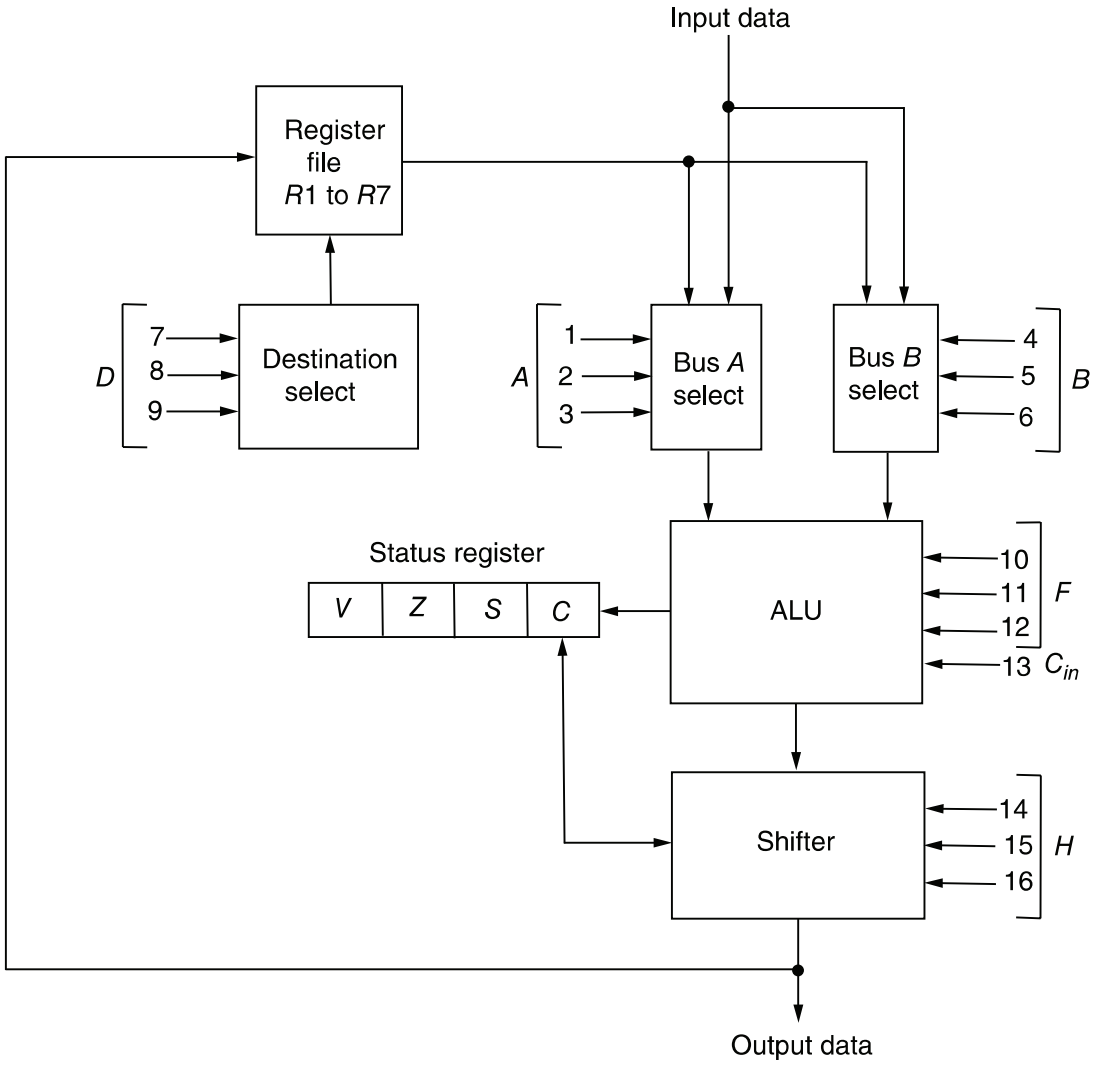
9.9 Processor Unit

The selection variables in a processor unit control the microoperations executed within the processor during any given clock pulse. The selection variables control the buses, the ALU, the shifter, and the destination register. We will now demonstrate by means of an example how the control variables select the microoperations in a processor unit. The example defines a processor unit together with all selection variables. Then we will discuss the choice of control variables for some typical microoperations.

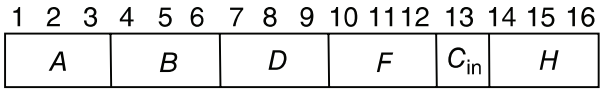
A block diagram of a processor unit is shown in Fig 9-16(a). It consists of seven registers $R1$ through $R7$ and a status register. The outputs of the seven registers go through two multiplexers to select the inputs to the ALU. Input data from an external source are also selected by the same multiplexers. The output of the ALU goes through a shifter and then to a set of external output terminals. The output from the shifter can be transferred to any one of the registers or to an external destination.

There are 16 selection variables in the unit, and their function is specified by a *control word* in Fig. 9-16(b). The 16-bit control word, when applied to the selection variables in the processor, specifies a given microoperation. The control word is partitioned into six fields, with each field designated by a letter name. All fields, except C_{in}, have a code of three bits. The three bits of A select a source register for the input to left side of the ALU. The B field is the same, but it selects the source information for the right input of the ALU. The D field selects a destination register. The F field, together with the bit in C_{in}, selects a function for the ALU. The H field selects the type of shift in the shifter unit.

The functions of all selection variables are specified in Table 9-8. The 3-bit binary code listed in the table specifies the code for each of the five fields A, B, D, F, and H. The register selected by A, B, and D is the one whose decimal number is equivalent to the binary number in the code. When the A or B field is 000, the corresponding multiplexer selects the input data. When $D = 000$, no destination register is selected. The three bits in the F field, together with the input carry C_{in}, provide the 12 operations of the ALU as specified in Table 9-4. Note that there are two possibilities for $F = A$. In one case the carry bit C is cleared, and in the other case it is set to 1 (see Table 9-2).



(a) Block diagram



(b) Control word

Figure 9-16 Processor unit with control variables

The first four entries for the code in the *H* field specify the shift operations of Table 9-7. A third selection variable is used to specify either a 0 for the serial inputs I_R and I_L or a circular shift with the carry bit C. For convenience, we designate a circular right-shift with carry by *crc* and a circular left-shift with carry, by *clc.* Thus, the statement:

$$R \leftarrow \text{crc } R$$

is an abbreviation for the statement:

$$R \leftarrow shr\ R, \qquad R_n \leftarrow C, \qquad C \leftarrow R_1$$

Table 9-8 Functions of control variables for the processor of Fig. 9-16

Binary code			Function of selection variables					
			A	B	D	F with $C_{in} = 0$	F with $C_{in} = 1$	H
0	0	0	Input data	Input data	None	$A, C \leftarrow 0$	$A + 1$	No shift
0	0	1	$R1$	$R1$	$R1$	$A + B$	$A + B + 1$	Shift-right, $I_R = 0$
0	1	0	$R2$	$R2$	$R2$	$A - B - 1$	$A - B$	Shift-left, $I_L = 0$
0	1	1	$R3$	$R3$	$R3$	$A - 1$	$A, C \leftarrow 1$	0's to output bus
1	0	0	$R4$	$R4$	$R4$	$A \vee B$	—	—
1	0	1	$R5$	$R5$	$R5$	$A \oplus B$	—	Circulate-right with C
1	1	0	$R6$	$R6$	$R6$	$A \wedge B$	—	Circulate-left with C
1	1	1	$R7$	$R7$	$R7$	$\overline{A}$	—	—

R is shifted to the right, its least significant bit R_1, goes to C, and the value of C goes into the most significant bit position R_n.

A control word of 16 bits is needed to specify a microoperation for the processor unit. The most efficient way to generate control words with so many bits is to store them in a memory unit which functions as a *control memory* where all control words are stored. The sequence of control words is then read from the control memory, one word at a time, to initiate the desired sequence of microoperations. This type of control organization is called *microprogramming* and is discussed in more detail in Chapter 10.

The control word for a given microoperation can be derived directly from the selection variables defined in Table 9-8. The subtract microoperation:

$$R1 \leftarrow R1 - R2$$

specifies $R1$ for the left input of the ALU, $R2$ for the right input of the ALU, $A - B$ for the ALU operation, no shift for the shifter, and $R1$ for the destination register. From Table 9-8, we derive the control word for this operation to be 0010100010101000:

A	B	D	F	C_{in}	H
001	010	001	010	1	000

The control words for this microoperation and a few others are listed in Table 9-9.

The *compare* operation is similar to the subtract microoperation, except that the difference is not transferred to a destination register; only the status bits are affected. The destination field D for this case must be 000. The transfer of $R4$ into $R5$ requires an ALU operation $F = A$. The source A is 100 and the destination D is 101. The B selection code could be anything because the ALU does not use it. This field is marked with 000 in the table for convenience, but any other 3-bit code could be used.

To transfer the input data into $R6$, we must have $A = 000$ to select the external input and $D = 110$ to select the destination register. Again the value of B does not matter and the ALU function

is $F = A$. To output data from $R7$, we make $A = 111$ and $D = 000$ (or 111). The ALU operation $F = A$ places the information from $R7$ into the output bus.

It is sometimes necessary to clear or set the carry bit before a circular-shift operation. This can be done with an ALU select code 0000 or 0111. With the first select code the C bit is cleared, and with the second code the C bit is set. The transfer $R1 \leftarrow R1$, $C \leftarrow 0$ does not change the contents of the register, but it clears C and V. The Z and S status bits are affected in the usual manner. If $R\ 1 = 0$, then Z is set to 1; otherwise, it is cleared. The S bit is set lo the value of the sign bit in $R1$.

The clock pulse that triggers the destination register also transfers the status bits from the ALU into the status register. The status bits are affected after the arithmetic operations. The C and V status bits are left unchanged during a logic operation, since these bits have no meaning for the logic operations. In some processors, it is customary not to change the value of carry bit C after an increment or decrement operation as well.

If we want to place the contents of a register into the shifter without changing the carry bit, we can use the OR logic operation with the same register selected for both ALU inputs A and B. The operation:

$$R \leftarrow R \vee R$$

does not change the value of register R. However, it does place the contents of R into the inputs of the shifter, and it *does not change* the values of status bits C and V.

The examples in Table 9-9 discussed thus far use the shift-select code 000 for the H field to indicate a no-shift operation. To shift the contents of a register, the value of the register must be placed into the shifter without any change through the ALU. The shift-left microoperation statement:

$$R3 \leftarrow \text{shl } R3$$

specifies the code for the shift select but not the code for the ALU. The contents of $R3$ can be placed into the shifter by specifying an OR operation between $R3$ and itself. The shifted

Table 9-9 Examples of microoperations for processor

	Control word						
Microoperation	A	B	D	F	C_{in}	H	Function
$R1 \leftarrow R1 - R2$	001	010	001	010	1	000	Subtract $R2$ from $R1$
$R3 - R4$	011	100	000	010	1	000	Compare $R3$ and $R4$
$R5 \leftarrow R4$	100	000	101	000	0	000	Transfer $R4$ to $R5$
$R3 \leftarrow$ Input	000	000	110	000	0	000	Input data to $R6$
Output $\leftarrow R7$	111	000	000	000	0	000	Output data from $R7$
$R1 \leftarrow R1$, $C \leftarrow 0$	001	000	001	000	0	000	Clear carry bit C
$R3 \leftarrow$ shl $R3$	011	011	011	100	0	010	Shift-left $R3$ with $I_L = 0$
$R1 \leftarrow$ crc $R1$	001	001	001	100	0	101	Circulate-right $R1$ with carry
$R2 \leftarrow 0$	000	000	010	000	0	011	Clear $R2$

information returns to $R3$ if $R3$ is specified as the destination register. This requires that select fields A, B, and D have the code 011 for $R3$, that the ALU function code be 1000 for the OR operation, and that the shift-select H be 010 for the shift-left.

The circular shift-right with carry of register $R1$ is symbolized by the statement:

$$R1 \leftarrow \text{crc } R1$$

This statement specifies the code for the shifter, but not the code for the ALU. To place the contents of $R3$ into the output terminals of the ALU without affecting the C bit, we use the OR operation as before. In this way, the C bit is not affected by the ALU operation but may be changed because of the circular shift.

The last example in Table 9-9 shows the control word for clearing a register to 0. To clear register $R2$, the output bus is made to contain al 0's, with $H = 011$. The destination field D is made equal to the code for register $R2$.

It is obvious from these examples that many more microoperations can be generated in the processor unit. A processor unit with a complete set of microoperations is a general-purpose device that can be adapted for many applicants. The register-transfer method is a convenient tool for specifying the operations in symbolic form in a digital system that employs a general-purpose processor unit. The system is first defined with a sequence of microoperation statements in the register-transfer method of notation or in any other suitable equivalent notation. A control function here is represented not by a Boolean function, but rather by a string of binary variables called a control word. The control word for each microoperation is derived from the function table of the processor

The sequence of control words for the system is stored in a control memory. The output of the control memory is applied to the selection variables of the processor. By reading consecutive control words from memory, it is possible to sequence the micro-operations in the processor. Thus, the entire design can be done by means of the register-transfer method which, in this particular case, is referred to as the *microprogramming method.* This method of controlling the processor unit is demonstrated in Section 10-5.

9.10 Design of Accumulator

Some processor units distinguish one register from all others and call it an accumulator register. The organization of a processor unit with an accumulator register is shown in Fig. 9-4. The ALU associated with the register may be constructed as a combinational circuit of the type discussed in Section 9-5. In this configuration, the accumulator register is essentially a bidirectional shift register with parallel load which is connected to an ALU. Because of the feedback connection from the output of the register to one of the inputs in the ALU, the accumulator register and its associated logic, when taken as one unit, constitute a sequential circuit. Because of this property, an accumulator register can be designed by sequential-circuit techniques instead of using a combinational-circuit ALU.

The block diagram of an accumulator that forms a sequential circuit is shown in Fig. 9-17, The A register and the associated combinational circuit constitute a sequential circuit. The combinational circuit replaces the ALU but cannot be separated from the register, since it is only the combinational-circuit part of a sequential circuit. The A register is referred to as the accumulator register and is sometimes denoted by the symbol AC. Here, accumulator refers to both the A reg-

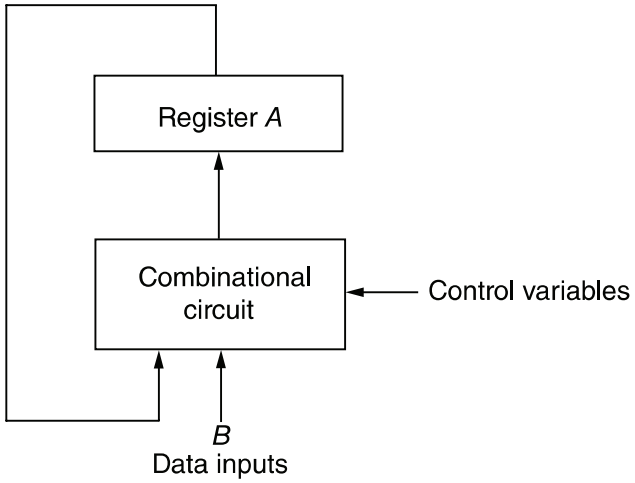


Figure 9-17 Block diagram of accumulator

ister and its associated combinational circuit. The external inputs to the accumulator are the data inputs from B and the control variables that determine the microoperations for the register. The next state of register A is a function of its present state and of the external inputs.

In Chapter 7, we considered various registers that perform specific functions such as parallel load, shift operations, and counting. The accumulator is similar to these registers but is more general, since it can perform not only the above functions, but also other data-processing operations. An accumulator is a multifunction register that, by itself, can be made to perform all of the microoperations in a processor unit. The microoperations included in an accumulator depend on the operations that must be included in the particular processor. To demonstrate the logic design of a multipurpose operational register such as an accumulator, we will design the circuit with nine microoperations. The procedure outlined in this section can be used to extend the register to other microoperations.

The set of microoperations for the accumulator is given in Table 9-10. Control variables p_1 through p_9 are generated by control logic circuits and should be considered as control functions

Table 9-10 List of microoperations for an accumulator

Control variable	Microoperation	Name
p_1	$A \leftarrow A + B$	Add
p_2	$A \leftarrow 0$	Clear
p_3	$A \leftarrow \overline{A}$	Complement
p_4	$A \leftarrow A \wedge B$	AND
p_5	$A \leftarrow A \vee B$	OR
p_6	$A \leftarrow A \oplus B$	Exclusive-OR
p_7	$A \leftarrow \text{shr } A$	Shift-right
p_8	$A \leftarrow \text{shl } A$	Shift-left
p_9	$A \leftarrow A + 1$	Increment
	If $(A = 0)$ then $(Z = 1)$	Check for zero

that initiate the corresponding register-transfer operations. Register A is a source register in all the listed microoperations. In essence, this represents the present state of the sequential circuit. The B register is used as a second source register for microoperations that need two operands. The B register is assumed to be connected to the accumulator and supplies the inputs to the sequential circuit. The destination register for all microoperations is always register A. The new information transferred to A constitutes the next state of the sequential circuit. The nine control variables are also considered as inputs to the sequential circuit. These variables are mutually exclusive and only one variable must be enabled when a clock pulse occurs. The last entry in Table 9-10 is a conditional control statement. It produces a binary 1 in an output variable Z when the content of register A is 0, i.e., when all flip-flops in the register are cleared.

9.10.1 Design Procedure

The accumulator consists of n stages and n flip-flops, A_1, A_2, …., A_n, numbered consecutively starting from the rightmost position. It is convenient to partition the accumulator into n similar stages, with each stage consisting of one flip-flop denoted by A_i, one data input denoted by B_i, and the combinational logic associated with the flip-flop. In the design procedure that follows, we consider only one typical stage i with the understanding that an n-bit accumulator consists of n stages for $i = 1, 2, ..., n$. Each stage A_i is interconnected with the neighboring stage A_{i-1} on its right and stage A_{i+1} on its left. The first stage, A_1, and the last stage, A_n, have no neighbors on one side and require special attention. The register will be designed using JK-type flip-flops.

Each control variable p_j, $j = 1, 2,..., 9$, initiates a particular microoperation. For the operation to be meaningful, we must ensure that only one control variable is enabled at any given time. Since the control variables are mutually exclusive, it is possible to separate the combinational circuit of a stage into smaller circuits, one for each microoperation. Thus, the accumulator is to be partitioned into n stages, and each stage is to be partitioned into those circuits that are needed for each microoperation. In this way, we can simplify the design process considerably. Once the various pieces are designed separately, it will be possible to combine them to obtain one typical stage of the accumulator and then to combine the stages into a complete accumulator.

Add B to A (p_1): The add microoperation is initiated when control variable p_1 is 1. This part of the accumulator can use a parallel adder composed of full-adder circuits as was done with the ALU. The full-adder in each stage i will accept as inputs the present state of A_i, the data input B_i, and a previous carry bit C_i. The sum bit generated in the full-adder must be transferred to flip-flop A_1, and the output carry C_{i+1}, must be applied to the input carry of the next stage.

The internal construction of a full-adder circuit can be simplified if we consider that it operates as part of a sequential circuit. The state table of a full-adder, when considered as a sequential circuit, is shown in Fig. 9-18. The value of flip-flop A_i before a clock pulse specifies the present state in the sequential circuit. The value of A_i after the application of a clock pulse specifies the next state. The next state of A_i is a function of its present state and inputs B_i and C_i. The present state and inputs in the state table correspond to the inputs of a full-adder. The next state and output C_{i+1} correspond to the outputs of a full-adder. But because it is a sequential circuit, A_i appears in both the present and next-state columns. The next state of A_i gives the sum bit that must be transferred to the flip-flop.

The excitation inputs for the JK flip-flop are listed in columns JA_i and KA_i. These values are obtained by the method outlined in Section 6-7. The flip-flop input functions and the Boolean

Present state	Inputs		Next state	Flip-flop inputs		Output
A_i	B_i	C_i	A_i	JA_i	KA_i	C_{i+1}
0	0	0	0	0	X	0
0	0	1	1	1	X	0
0	1	0	1	1	X	0
0	1	1	0	0	X	1
1	0	0	1	X	0	0
1	0	1	0	X	1	1
1	1	0	0	X	1	1
1	1	1	1	X	0	1

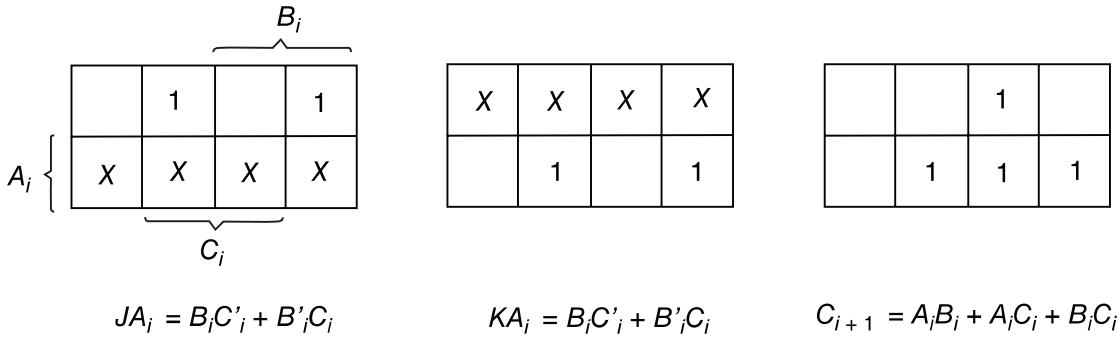


Figure 9-18 Excitation table for add microoperation

function for the output are simplified in the maps of Fig. 9-18. The *J* input of flip-flop A_i, designated by JA_i, and the *K* input of flip-flop A_i, designated by KA_i, do not include the control variable p_1. These two equations should affect the flip-flop only when p_1 is enabled; therefore, they should be ANDed with control variable p_1. The part of the combinational circuit associated with the add microoperation can be expressed with three Boolean functions:

$$JA_i = B_iC'_ip_1 + B'_iC_ip_1$$
$$KA_i = B_iC'_ip_1 + B'_iC_ip_1$$
$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

The first two equations are identical, and they specify a condition for complementing A_i. The third equation generates the carry for the next stage.

Clear (p_2): Control variable p_2 clears all flip-flops in register *A*. To cause this transition in a *JK* flip-flop, we need only apply control variable p_2 *to* the *K* input of the flip-flop. The *J* input

will be assumed to be 0 if nothing is applied to it. The input functions for the clear microoperation are:

$$JA_i = 0$$
$$KA_i = p_2$$

Complement (p_3): Control variable p_3 complements the state of register A. To cause this transition in a JK flip-flop, we need to apply p_3 to both the J and K inputs:

$$JA_i = p_3$$
$$KA_i = p_3$$

AND (p_4): The AND microoperation is initiated with control variable p_4. This operation forms the logic AND operation between A_i and B_i and transfers the result to A_i. The excitation table for this operation is given in Fig. 9-19(a). The next state of A_i is 1 only when both B_i and the present state of A_i are equal to 1. The flip-flop input functions which are simplified in the two maps dictate that the K input of the flip-flop be enabled with the complement value of B_i. This result can be verified from the conditions listed in the state table. If $B_i = 1$, the present state and next state of A_i are the same, so the flip-flop does not have to undergo a change of state. If $B_i = 0$, the next state of A_i must go to 0, and this is accomplished by enabling the K input of the flip-flop. The input functions for the AND microoperation must include the control variable that initiates this microoperation:

$$JA_i = 0$$
$$KA_i = B'_i p_4$$

OR (p_5): Control variable p_5 initiates the logic OR operation between A_i and B_i, with the result transferred to A_i. Figure 9-19(b) shows the derivation of the flip-flop input functions for this operation. The simplified equations in the maps dictate that the J input be enabled when $B_i = 1$. This result can be verified from the state table. When $B_i = 0$, the present state and next state of A_i are the same. When $B_i = 1$, they input is enabled and the next state of A_i becomes 1. The input functions for the OR microoperation are:

$$JA_i = B_i p_5$$
$$KA_i = 0$$

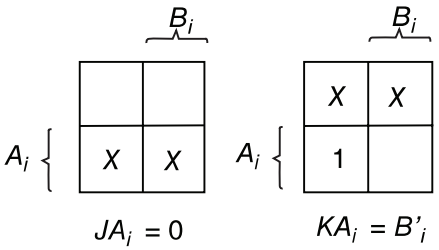
Exclusive-OR (p_6): This operation forms the logic exclusive-OR between A_i and B_i and transfers the result to A_i. The pertinent information for this operation is shown in Fig. 9-19(c). The flip-flop input functions are:

$$JA_i = B_i p_6$$
$$KA_i = B_i p_6$$

Shift-right (p_7): This operation shifts the contents of the A register one position to the right. This means that the value of flip-flop A_{i+1}, which is one position to the left of stage i, must be

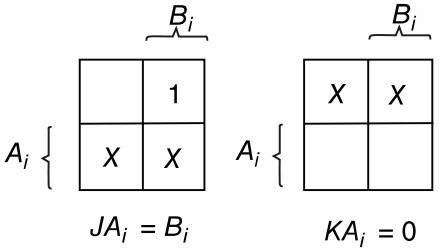
Present state	Input	Next state	Flip-flop inputs	
A_i	B_i	A_i	JA_i	KA_i
0	0	0	0	X
0	1	0	0	X
1	0	0	X	1
1	1	1	X	0

(a) AND



Present state	Input	Next state	Flip-flop inputs	
A_i	B_i	A_i	JA_i	KA_i
0	0	0	0	X
0	1	1	1	X
1	0	1	X	0
1	1	1	X	0

(b) OR



Present state	Input	Next state	Flip-flop inputs	
A_i	B_i	A_i	JA_i	KA_i
0	0	0	0	X
0	1	1	1	X
1	0	1	X	0
1	1	0	X	1

(c) Exclusive-OR

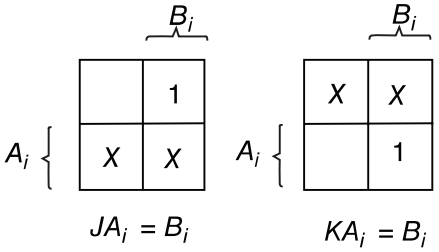


Figure 9-19 Excitation tables for logic microoperations

transferred into flip-flop A_i. This transfer is expressed by the input functions:

$$JA_i = A_{i+1}p_7$$
$$KA_i = A'_{i+1}p_7$$

Shift-left (p_8): This operation shifts the A register one position to the left. For this case, the value of A_{i-1}, which is one position to the right of stage i, must be transferred to A_i. This transfer is expressed by the input functions:

$$JA_i = A_{i-1}p_8$$
$$KA_i = A'_{i-1}p_8$$

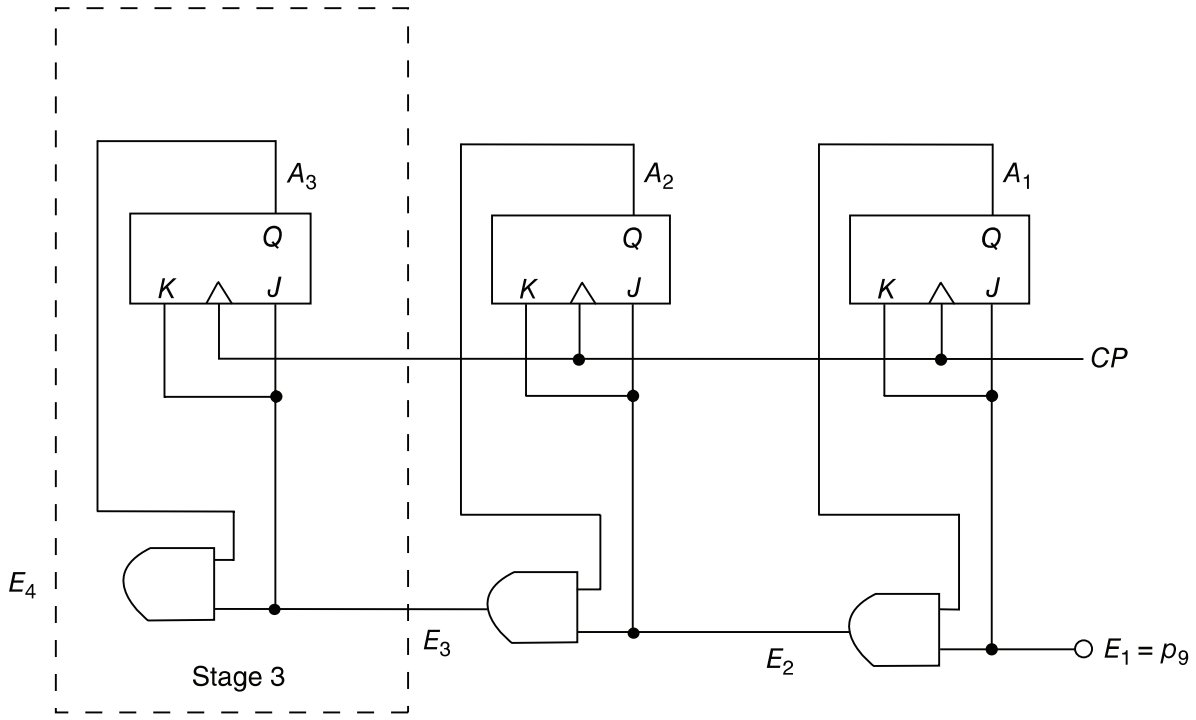


Figure 9-20 3-bit synchronous binary counter

Increment (p_9): This operation increments the contents of the A register by one; in other words, the register behaves like a synchronous binary counter with p_9 enabling the count. A 3-bit synchronous counter is shown in Fig. 9-20. It is similar to the counter in Fig. 7-17 of Section 7-5, where the operation of synchronous binary counters are discussed in detail. From the diagram, we see that each stage is complemented when an input carry $E_i = 1$. Each stage also generates an output carry, E_{i+1}, for the next stage on its left. The first stage is an exception, since it is complemented with the count-enable p_9. The Boolean functions for a typical stage can be expressed as follows:

$$\begin{aligned} JA_i &= E_i \\ KA_i &= E_i \\ E_{i+1} &= EA_i \qquad\qquad i = 1, 2, \ldots, n \\ E_i &= p_9 \end{aligned}$$

The input carry, E_i, into the stage is used to complement flip-flop A_i. Each stage generates a carry for the next stage by ANDing the input carry with A_i. The input carry into the first stage is E_1, and must be equal to control variable p_9 which enables the count.

Check for Zero (Z): Variable Z is an output from the accumulator used to indicate a zero content in the A register. This output is equal to binary 1 when all the flip-flops are cleared. When a flip-flop is cleared, its complement output, Q', is equal to 1. Figure 9-21 shows the first three stages of the accumulator that checks for a *zero* content. Each stage generates a variable z_{i+1} by ANDing the complement output of A_i to an input variable z_i. In this way, a chain of AND gates through all stages will indicate if all flip-flops are cleared. The Boolean functions for a typical

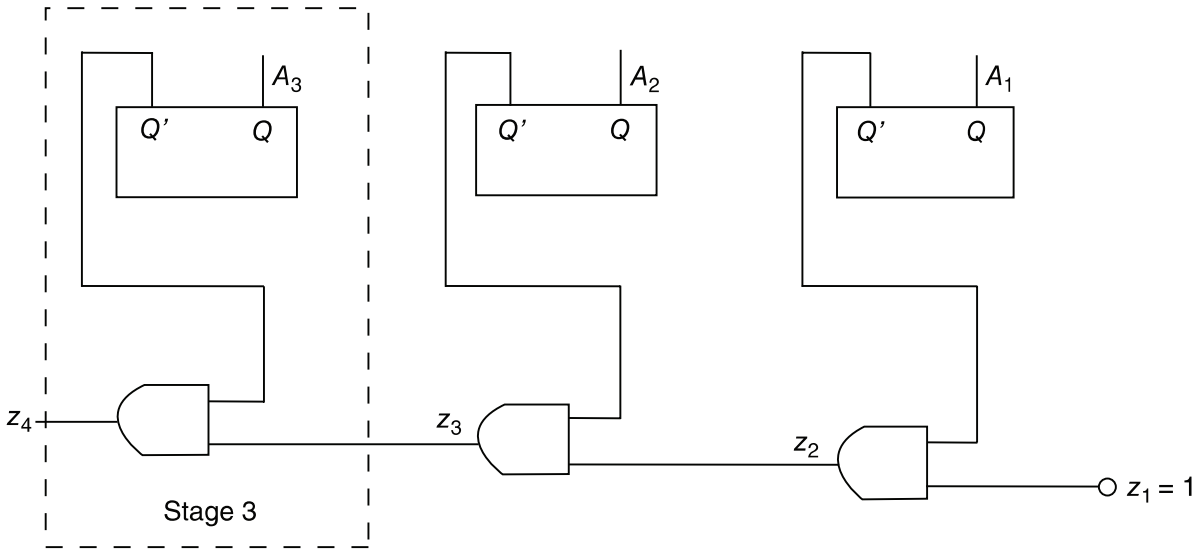


Figure 9-21 Chain of AND gates for checking the zero content of a register

stage can be expressed as follows:

$$z_{i+1} = z_1 A'_i \qquad i = 1, 2, \ldots, n$$
$$z_i = 1$$
$$z_{n+1} = Z$$

Variable Z becomes 1 if the output signal from the last stage, z_{n+1}, is 1.

9.10.2 One Stage of Accumulator

A typical accumulator stage consists of all the circuits that were derived for the individual microoperations. Control variables p_1 through p_9 are mutually exclusive; therefore, the corresponding logic circuits can be combined with an OR operation. Combining all the input functions for the J and the K inputs of flip-flop A_i produces a composite set of input Boolean functions for a typical stage:

$$JA_i = B_i C'_i p_1 + B'_i C_i p_1 + p_3 + B_i p_5 + B_i p_6 + A_{i+1} p_7 + A_{i-1} p_8 + E_i$$
$$KA_i = B_i C'_i p_1 + B'_i C_i p_1 + p_2 + p_3 + B'_i p_4 + B_i p_6 + A'_{i+1} p_7 + A'_{i-1} p_8 + E_i$$

Each stage in the accumulator must also generate the carries for the next stage:

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$$
$$E_{i+1} = E_i A_i$$
$$z_{i+1} = z_i A'_i$$

The logic diagram of one typical stage of the accumulator is shown in Fig. 9-22. It is a direct implementation of the Boolean functions listed above. The diagram is a composite circuit that includes the individual circuits associated with each microoperation. The various circuits are combined with two OR gates in the J and K inputs of flip-flop A_i.

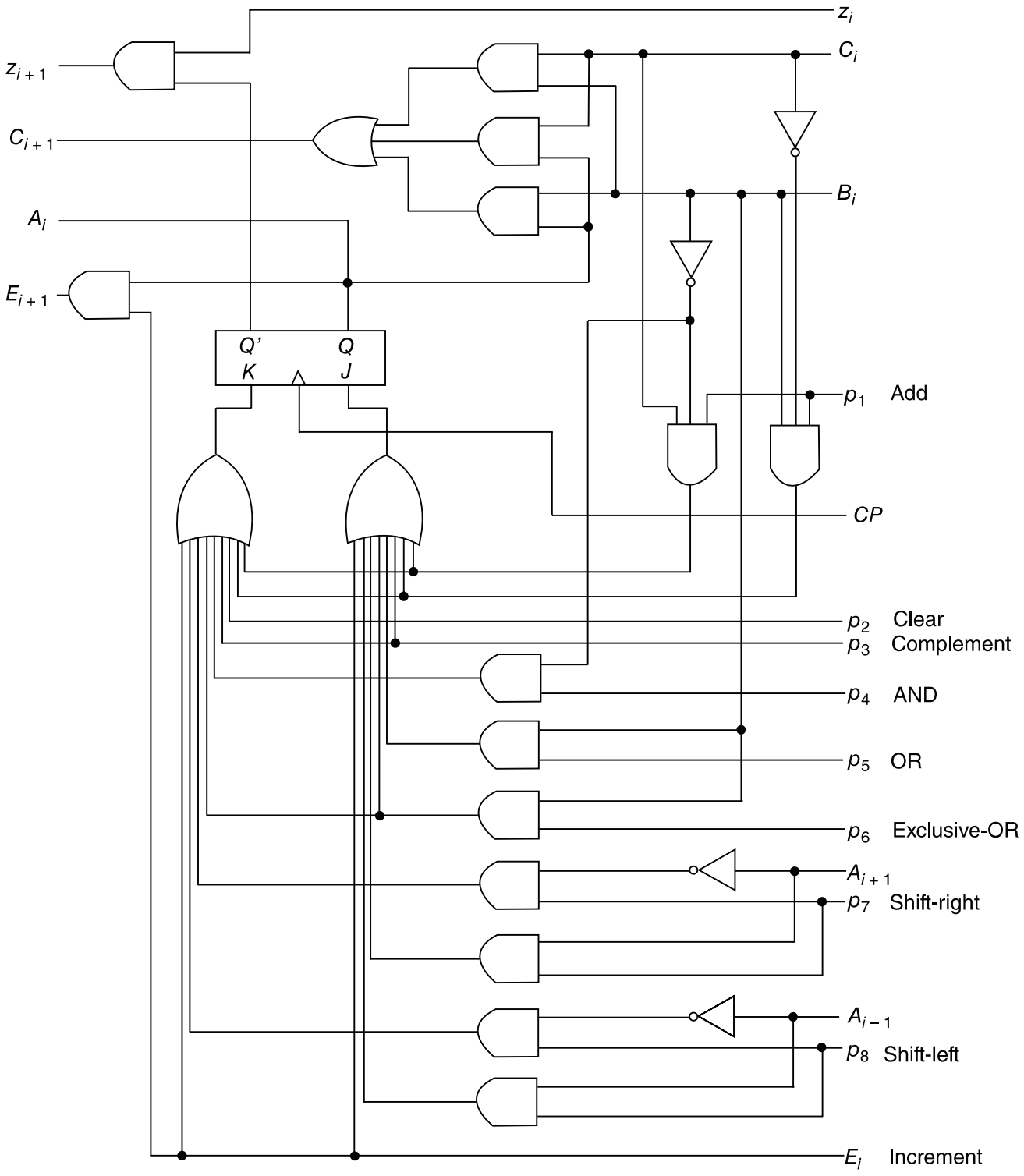


Figure 9-22 One typical stage of the accumulator

Each accumulator stage has eight control inputs, p_1 through p_8, that initiate one of eight possible microoperations. Control variable p_9 is applied only to the first stage to enable the increment operation through input E_i. There are six other inputs in the circuit. B_i is the data bit from the B terminals that provide the inputs to the accumulator. C_i is the input carry from the previous stage on the right. A_{i-1} comes from the output of the flip-flop one position to the right, and A_{i+1}

comes from the flip-flop one position to the left, E_i is the carry input for the increment operation, and z_i is used to form the chain for zero detection. The circuit has four outputs: A_i is the output of the flip-flop, C_{i+1} is a carry for the next stage, E_{i+1} is the increment carry for the next stage, and z_{i+1} is for the next stage on the left to form the chain for zero detection.

9.10.3 Complete Accumulator

An accumulator with n bits requires n stages connected in cascade, with each stage having the circuit shown in Fig. 9-22. All control variables, except p_9, must be applied to each stage. The other inputs and outputs in each stage must be connected in cascade to form a complete accumulator.

The interconnection among stages to form a complete accumulator is illustrated in the 4-bit accumulator shown in Fig. 9-23. Each block in the diagram represents the circuit of Fig. 9-22. The number on top of each block represents the bit position in the accumulator. All blocks receive eight control variables, p_1 through p_8, and the clock pulses from *CP*. The other six inputs and four outputs in each block are identical to those of a typical stage, except that subscript i is now replaced by the particular number in each block.

The circuit has four B inputs. The zero-detect chain is obtained by connecting the z variables in cascade, with the first block receiving a binary constant 1. The last stage in this chain produces the zero-detect variable Z. The carries for the arithmetic addition are connected in cascade as in full-adder circuits. The serial input for the shift-left operation goes to input A_0, which corresponds to input A_{i-1} in the first stage. The serial input for the shift-right operation goes to input A_5, which corresponds to A_{i+1} in the fourth and last stage. The increment operation

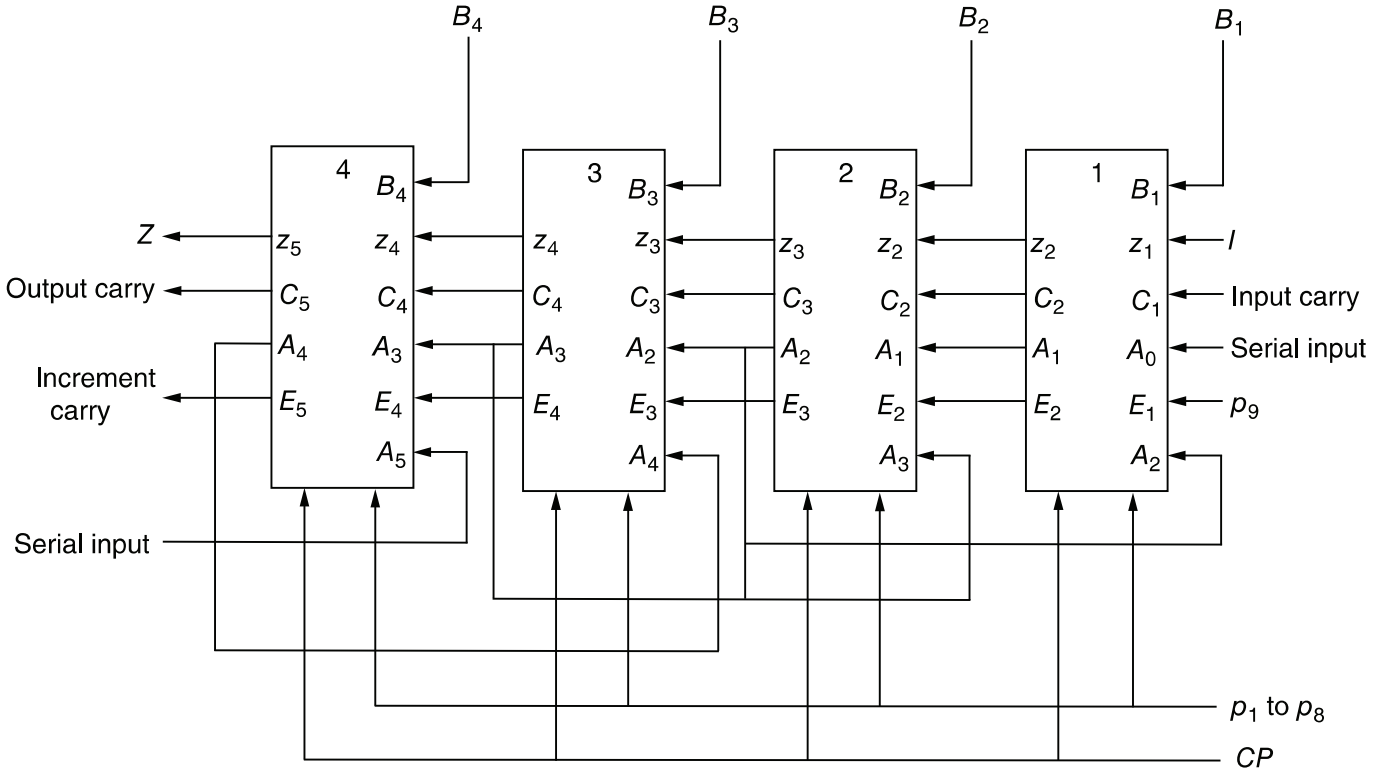


Figure 9-23 4-bit accumulator constructed with four stages

is enabled with control variable p_9 in the first stage. The other blocks receive the increment carry from the previous stage.

The total number of terminals in the 4-bit accumulator is 25, including terminals for the *A* outputs. Incorporating two more terminals for power supply, the circuit can be enclosed within one IC package having 27 or 28 pins. The number of terminals for the control variables can be reduced from nine to four if a decoder is inserted in the IC. In such a case, the IC pin count can be reduced to 22 and the accumulator can be extended to 16 microoperations without adding external pins.

REFERENCES

1. Mano, M. M., *Computer System Architecture.* Englewood Cliffs, NJ.: Prentice-Hall, Inc., 1976.
2. *The TTL Data Book for Design Engineers.* Dallas, Texas: Texas Instruments, Inc., 1976.
3. The *Am*2900 *Bipolar Microprocessor Family Data Book.* Sunnyvale, Calif.: Advanced Micro Devices, Inc., 1976.
4. Sobel, H. S., *Introduction to Digital Computer Design.* Reading. Mass.: Addison-Wesley Publishing Co., 1970.
5. Kline, R. M., *Digital Computer Design.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1977.
6. Chirlian, P. M., *Analysis and Design of Digital Circuits and Computer Systems.* Champaign. Ill.: Matrix Publishing, Inc., 1976.

PROBLEMS

9-1. Modify the processor unit of Fig. 9-1 so that the selected destination register is always the same register that is selected for the *A* bus. How does this effect the number of multiplexers and the number of selection lines used?

9-2. A bus-organized processor as in Fig. 9-1 consists of 15 registers. How many selection lines are there in each multiplexer and in the destination decoder?

9-3. Assume that each register in Fig. 9-1 is 8 bits long. Draw a detailed block diagram for the box labeled MUX that selects the register for the *A* bus. Show that the selection can be done with eight 4-to-1 line multiplexers.

9-4. A processor unit employs a scratchpad memory as in Fig. 9-2. The processor consists of 64 registers of eight bits each.

(a) What is the size of the scratchpad memory?

(b) How many lines are needed for the address?

(c) How many lines are there for the input data?

(d) What is the size of the MUX that selects between the input data and the output of the shifter?

9-5. Show a detailed block diagram for the processor unit of Fig. 9-4 when the *B* inputs come from:

(a) Eight processor registers forming a bus system.

(b) A memory unit with address and buffer registers.

9-6. The 4-bit ALU of Fig. 9-5 is enclosed within one IC package. Show the connections among three such ICs to form a 12-bit ALU. Designate the input and output carries in the 12-bit ALU.

9-7. TTL IC type 7487 is a 4-bit true /complement, zero/one element. One stage of this IC is shown in Fig. P9-7.