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 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 applications. 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 microoperations 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 register and its associated combinational circuit. The external inputs to the accumulator are the data inputs from B and the control variables that determine the

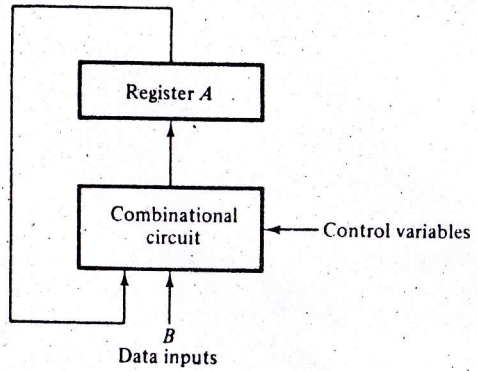


Figure 9-17 Block diagram of accumulator

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 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.

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

Design Procedure

The accumulator consists of n stages and n flip-flops, $A_1, A_2, \ldots, A_n$, 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 $i = 1, 2, \ldots, n$. 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.

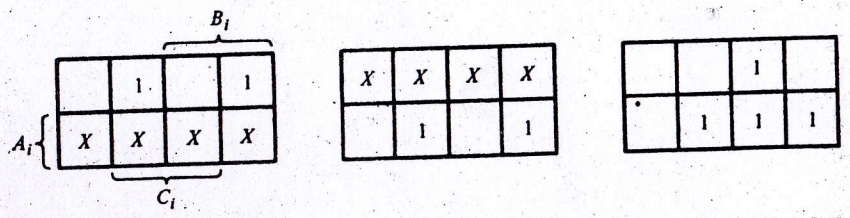
Each control variable p_j, $j = 1, 2, \ldots, 9$, initiates a particular microoperation. For the operation to be meaningful, we must ensure that only one control variable is enabled at any given time. Since the control variables are mutually exclusive, it is possible to separate the combinational circuit 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_i, 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 ful-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 function for the output are simplified in the maps

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



$JA_i = B_iC_i' + B_i'C_i$ $\quad$ $KA_i = B_iC_i' + B_i'C_i$ $\quad$ $C_{i+1} = A_iB_i + A_iC_i + B_iC_i$

Figure 9-18 Excitation table for add microoperation

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:

$$JA_i = B_iC_i'p_1 + B_i'C_ip_1$$
$$KA_i = B_iC_i'p_1 + B_i'C_ip_1$$
$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

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:

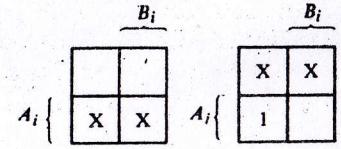
$$JA_i = 0$$
$$KA_i = p_2$$

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:

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

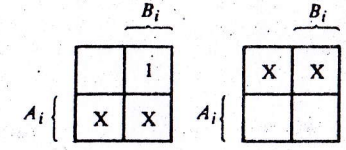
AND (p_4): The AND microoperation is initiated with control variable p_4. This operation forms the logic AND operation between A_i and B_i and transfers the result to A_i. The excitation table for this operation is given in Fig. 9-19(a). The next state of A_i is 1 only when both B_i and the present state of A_i are equal to 1. The flip-flop input functions which are simplified in the two maps dictate that the

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



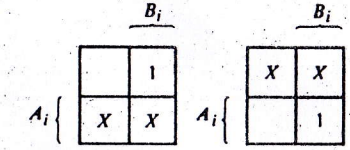
(a) AND

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

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 $B_i = 1$, the J 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 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:

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

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 a enabling the count. A 3-bit synchronous counter is shown in Fig. 9-20. It is simi

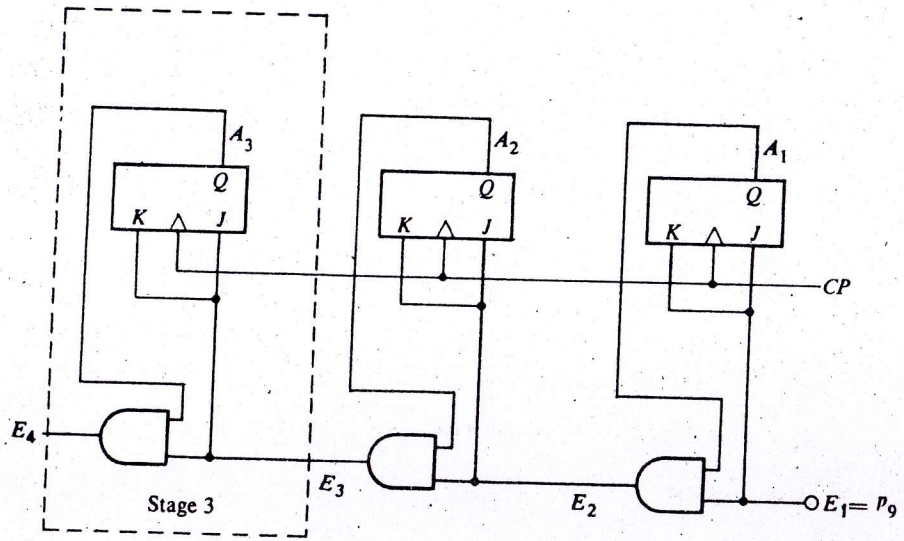


Figure 9-20 3-bit synchronous binary counter

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:

$$JA_i = E_i$$
$$KA_i = E_i$$
$$E_{i+1} = E_i A_i \qquad i = 1, 2, \ldots, n$$
$$E_1 = p_9$$

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

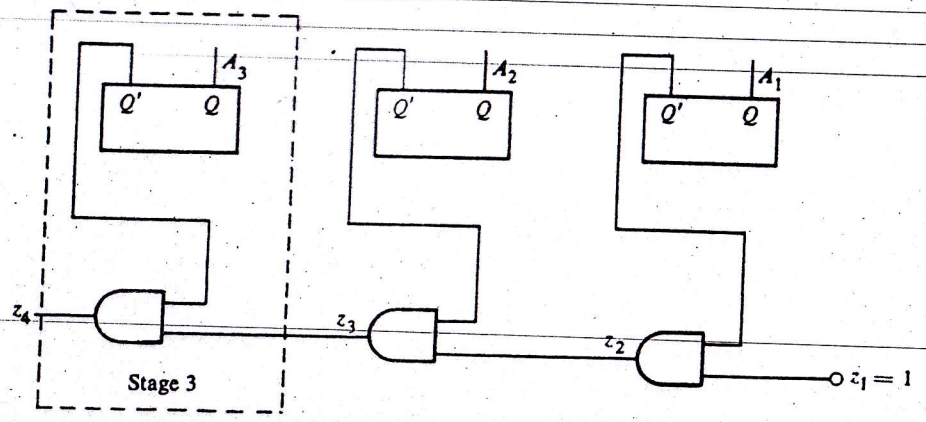


Figure 9-21 Chain of AND gates for checking the zero content of a register

for a typical stage can be expressed as follows:

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

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

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

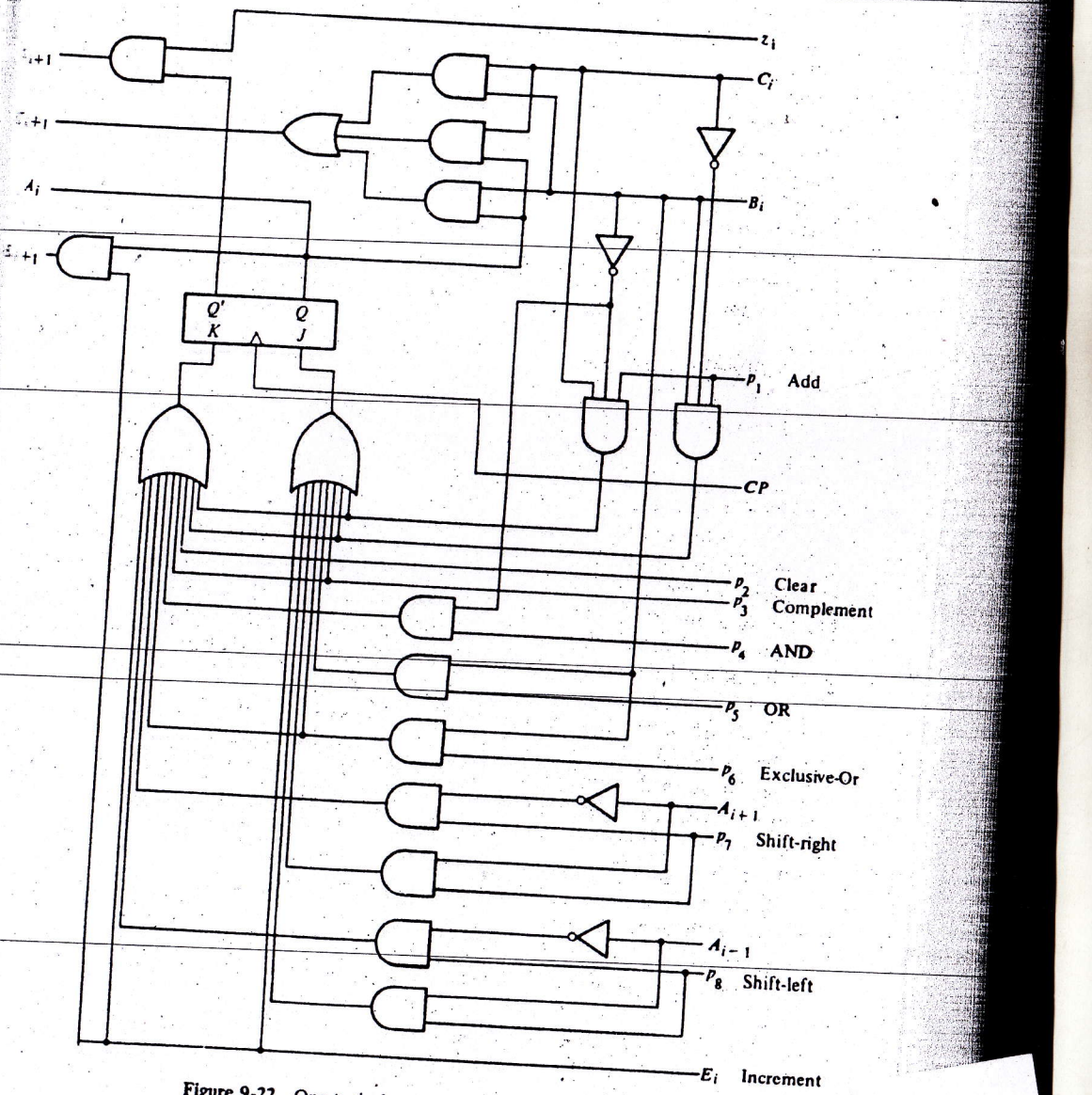


Figure 9-22 One typical stage of the accumulator