Module - 2. Combinational Logic:

* Introduction: -

* A logic circuit is combinational if its o/ps at any time are a function of only the present inputs.

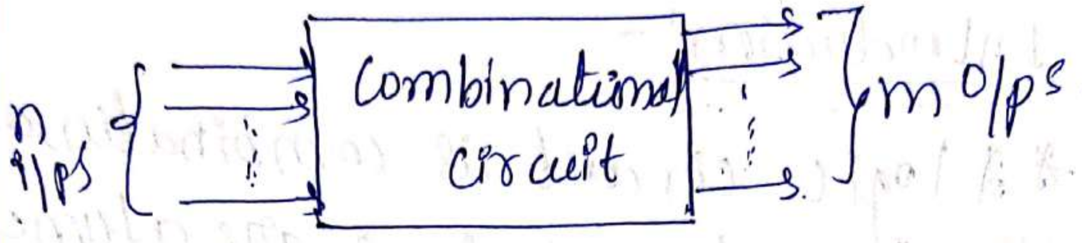
eg: All basic gates AND. OR. NOT. and also Decoder; encoder. mux. Demux etc.

* Combinational circuits:-

* combinational circuit consists of an interconnection of logic gates.

* combinational logic circuit will transfer the binary information from the given input data to a required output data.

* A block diagram of a combinational circuit is shown in fig(1).



fig① Block diagram of combinational circuit.

* The n input binary variables come from an external source.
* the m output variables are produced by the input signals acting on the internal combinational logic circuit & go to an external destination.
* For n inputs variables there are 2^n possible combinations of the binary inputs.
* For each possible i/p combination, there is one possible value for each o/p variable.
* Thus, a combinational circuit can be specified with a truth table that lists

the output values for each combination of input variables.

* Design procedure:-

* The design of combination circuits starts from the specification of the design objective or a set of Boolean f^ns. from which the logic diagrams can be obtained

* The procedure involves the following steps.

1) From the specifications of the circuit, determine the required number of i/ps and outputs & assign a symbol to each.

2) Derive the truth table that defines the required relationship b/w inputs and outputs.

3) obtain the simplified Boolean f^ns for each o/ps as a f^n of the i/p Variable.

4) Draw. the logic · diagram and verify the · correctness · of the design (manually or by simulation)

* <u>Code. Conversion Example</u> :-

* A conversion · circuit must be inserted between the two systems if each · uses different · codes · for the same - information

* Thus. a code converter is · a circuit that makes · the two systems compatible · even though each. uses a different binary code let us consider example · that converts BCD · to the excess-3 · code for the decimal digits.

```
BCD    0000          BCD 1001
      +0011             +0011
      -----             -----
Excess-3 0011            1100
```

Truth table for code conversion Example

I/p BCD — O/P Excess-3 code

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

For Z

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	1_0	0_1	0_3	1_2
$\bar{A}B$ 01	1_4	0_5	0_7	1_6
AB 11	X_{12}	X_{13}	X_{15}	X_{14}
$A\bar{B}$ 10	1_8	0_9	X_{11}	X_{10}

$$\boxed{Z = \bar{D}}$$

For Y

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	1	0	1	0
$\bar{A}B$ 01	1	0	1	0
AB 11	X	X	X	X
$A\bar{B}$ 10	1	0	X	X

$$Y = \bar{C}\bar{D} + CD.$$

For X

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	0	1	1	1
$\bar{A}B$ 01	1	0	0	0
AB 11	X	X	X	X
$A\bar{B}$ 10	0	1	X	X

~~2~~ $X = B\bar{C}\bar{D} + \bar{B}D + \bar{B}\bar{D}$

For W

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	0	0	0	0
$\bar{A}B$ 01	0	1	1	1
AB 11	X	X	X	X
$A\bar{B}$ 10	1	1	X	X

$X = A + BD +$ ~~BD~~ BC

The expression obtained from each k-map may be manipulated algebraically for the purpose of using common gates for two or more outputs.

* The manipulation give flexibility to obtained with the multiple-output systems. when implemented with three or more level of gates.

$z = \overline{D}$

$y = CD + \overline{C}\,\overline{D} = CD + \overline{(C+D)}$

$x = \overline{B}C + \overline{B}D + B\overline{C}\,\overline{D} = \overline{B}\,(C+D) + B\overline{C}\,\overline{D}$

$x = \overline{B}\,(C+D) + B\overline{(C+D)}$

$w = A + BC + BD = A + B(C+D)$

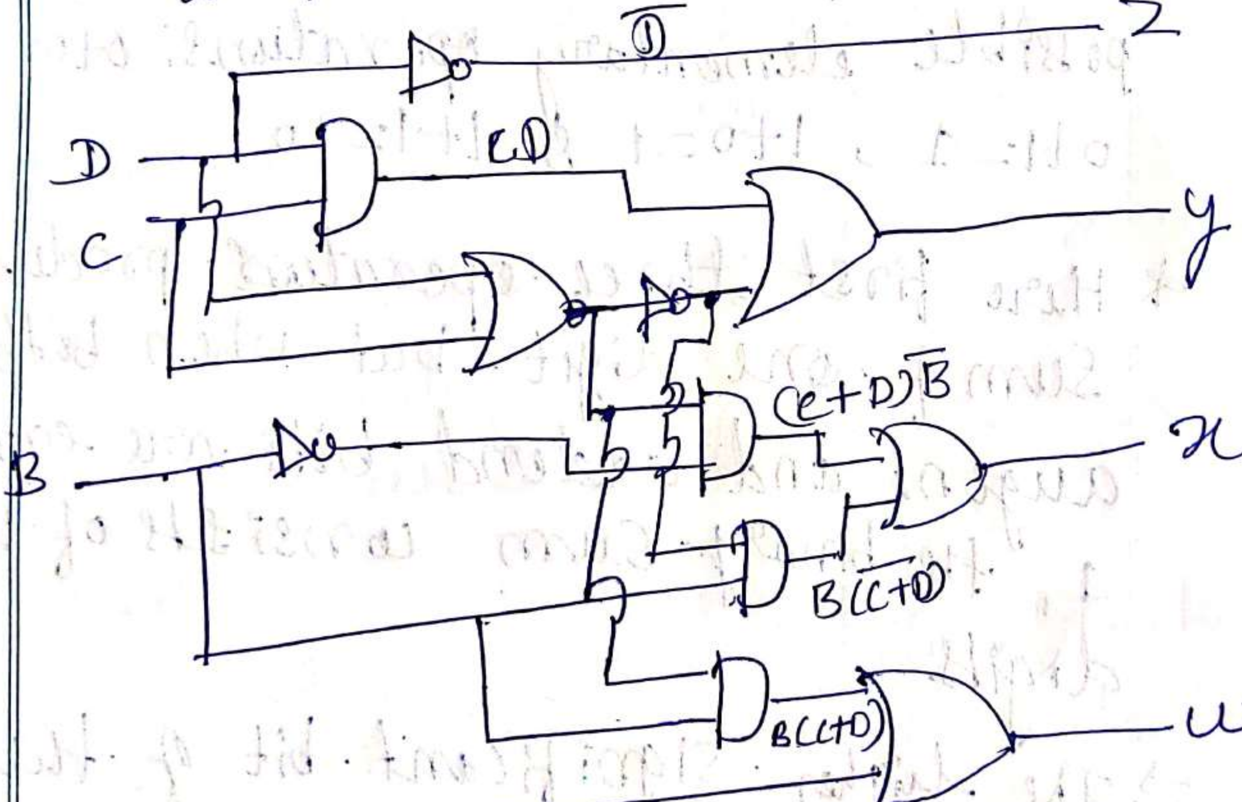


fig 2 Logic diagram. for BCD-to Excess-3 code converter.

* Binary Adder - Subtractor:-

* Digital computers perform a variety of information-processing tasks.

* Among the functions encountered are the various arithmatic operations.

* The most basic arithmetic operation is the addition of two binary digits.

* This simple addition consists of four possible elementary operations: 0+0=0 0+1=1 , 1+0=1 & 1+1=10.

* Here first ~~three~~ operations produce a sum of one digit but when both augend and addend bits are equal to 1, the binary sum consists of two digits.

→ The higher significant bit of this result is called a carry.

* A combination circuit. ~~of~~ that performs the addition of two bits. is called a half adder.

* one that perform the addition of three bits. is a full-adder.

* Half Adder:-
* From the verbal explanation of a HA we find that this circuit needs two binary I/ps & two binary O/ps.
* The i/p variables designate the augend & addend bits.
* The o/p variables produce the sum & carry.
* let us assign symbol x & y to the two I/ps & S (for sum) & C (for carry). to the o/ps.

T.T. HA.

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

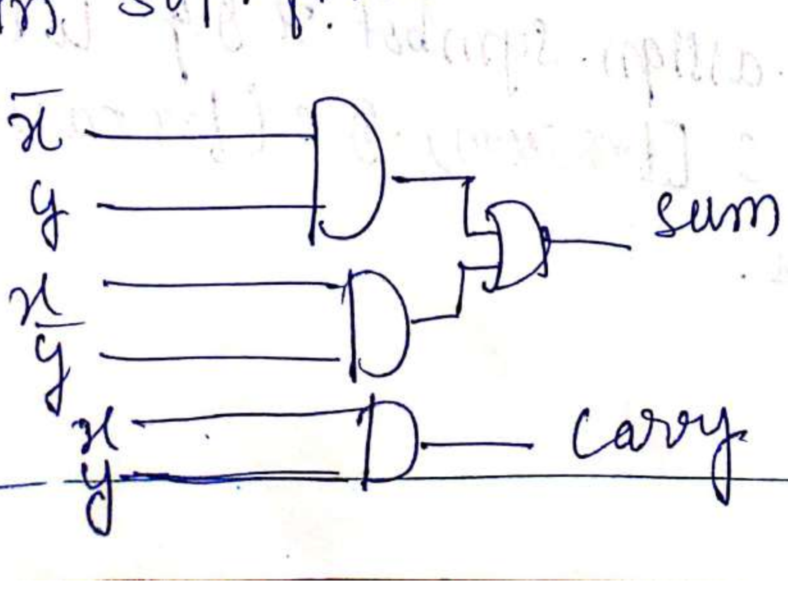
* The C O/P is 1 only when both. I/Ps are 1
The S O/P represents ~~the least~~ sum.

* The simplified sum-of-products expressions are

$$S = \bar{x}y + x\bar{y}$$

$$C = xy$$

logic diagram of the HA. implemented in SOP. form is shown in fig 1.



$S = x \oplus y \quad c = xy$

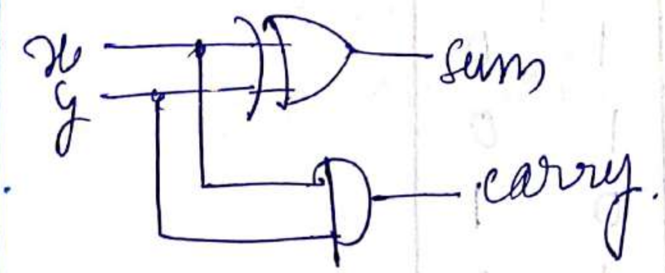


fig (b) Implementation of HA

* Full Adder

The process of addition proceeds on a bit-by-bit basis, right to left. begining with the least significant bit.

* A FA is a combinational circuit that forms the arithmetic sum of three bits.

* It consists of three i/ps & two o/ps.

T.T. for FA:-

x	y	z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

For sum:

x \ yz	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$\text{sum} = \bar{x}\bar{y}z + \bar{x}y\bar{z} + x\bar{y}\bar{z} + xyz.$

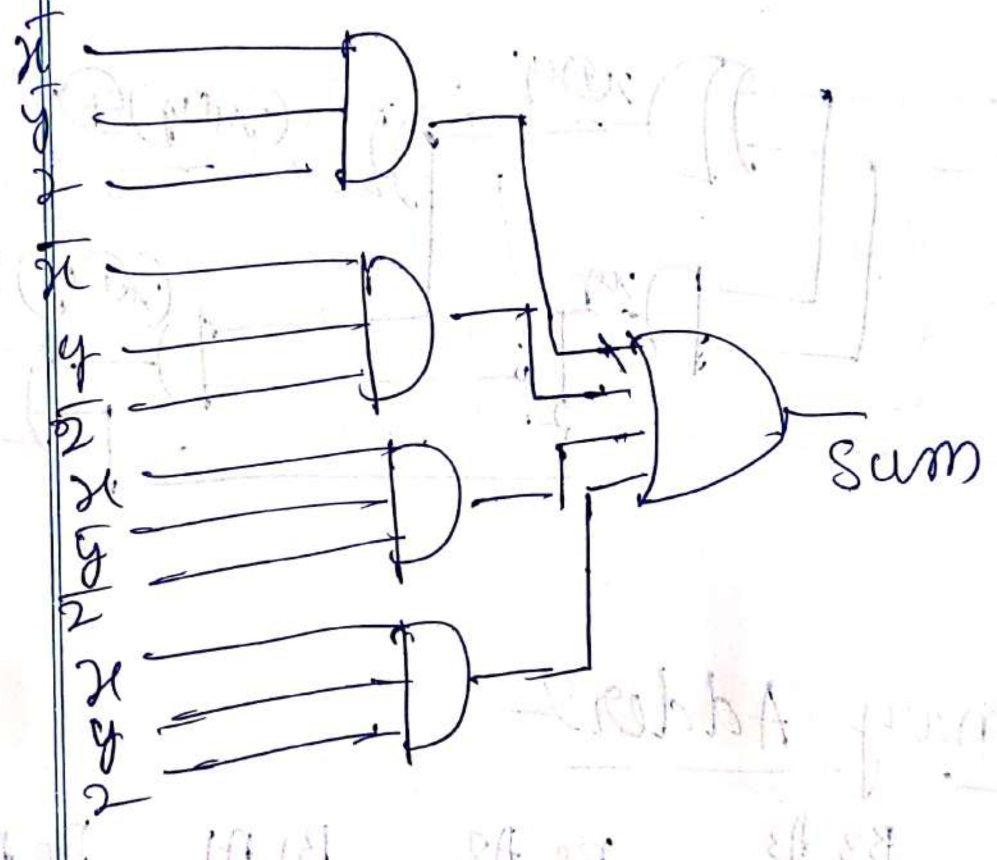
For carry

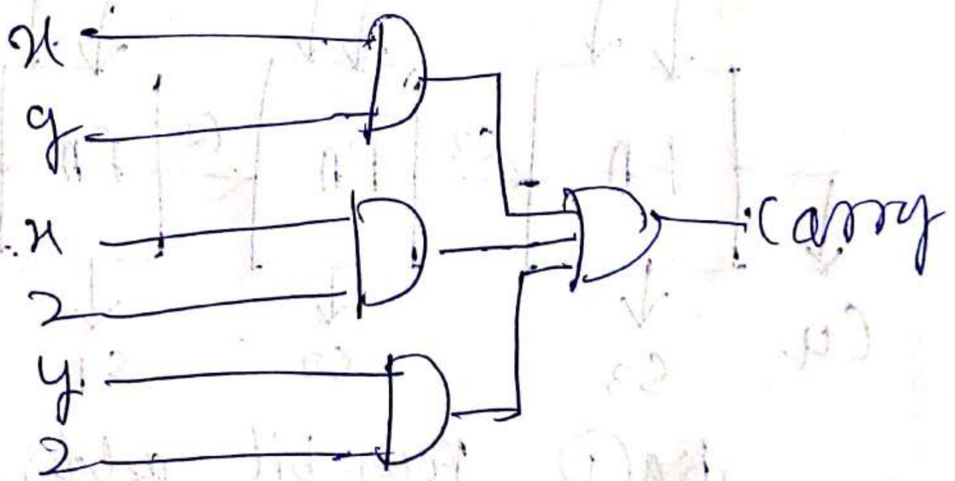
x \ yz	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$\text{carry} = yz + xz + xy$

The logical diagram for the FA implemented in Sop. form shown in fig③.

* It can also implemented with two HA & one OR gate as shown in fig④

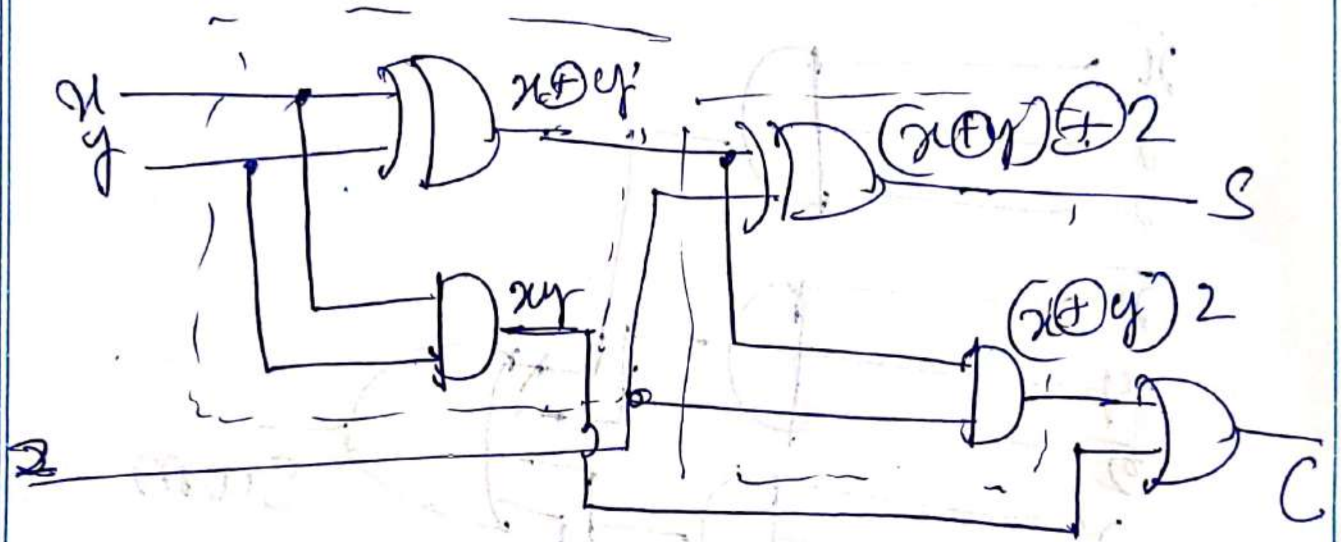




$S = z \oplus (x \oplus y)$

$\text{Carry} = z(x\bar{y} + \bar{x}y) + xy$

$= x\bar{y}z + \bar{x}yz + xy$



Binary Adder

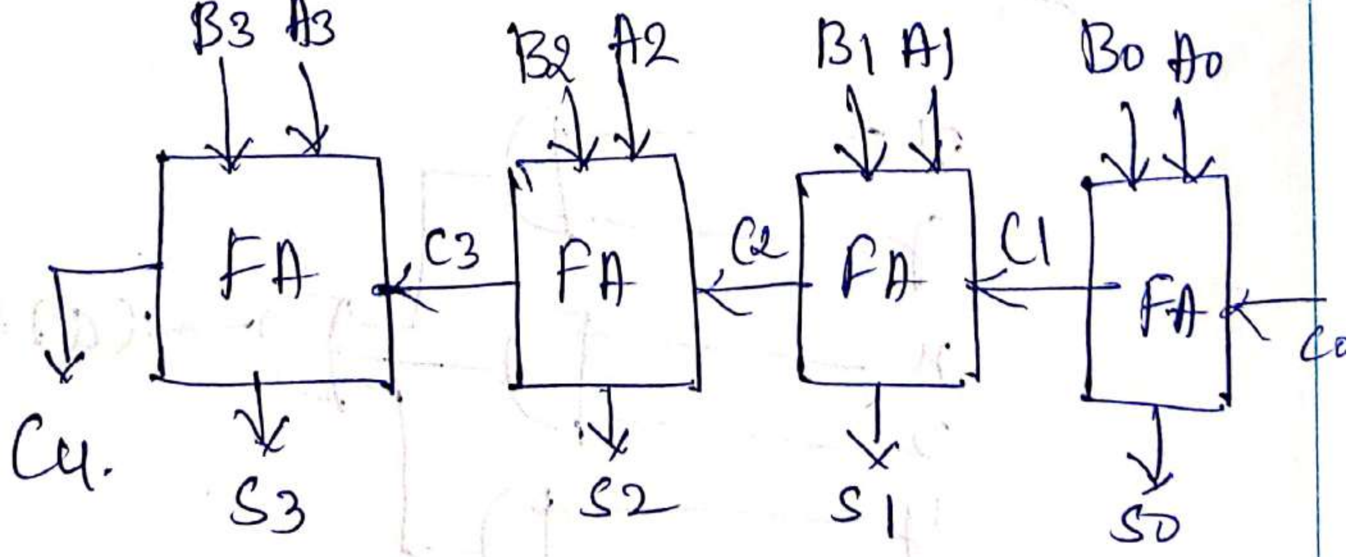


fig ① Four-bit adder.

fig① shows the connection of four full-adder (FA) ckt to provide a four-bit binary ripple carry adder.

* Let consider eg. $\overset{A_3A_2A_1A_0}{A=1011}$ $\overset{B_3B_2B_1B_0}{B=0011}$

Sum = 1110 is formed.

$C_1 = 0$

Subscript i.	3	2	1	0	
I/p carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
O/p carry	0	0	1	1	C_{i+1}

Carry propagation :-

* Addition of two binary numbers in parallel implies that all the bits of the augend & addend are available for computation at the same time.

* The total propagation time is equal to the delay of a typical gate, times the number of gates levels in the circuit.

* The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adder.

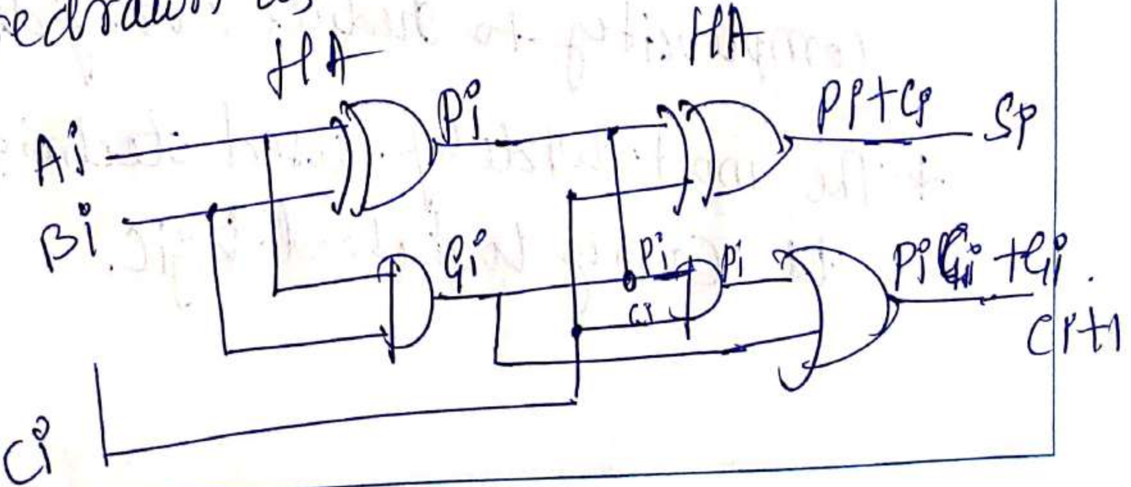
* Since each bit of the sum O/p depends on the value of the I/p carry.

* Si at any given stage in all adder ~~will be~~ to get final value only after the I/p carry to that stage has been propagated.

* to get S_3 value, it depends on the I/P A_3, B_3 & C_3. but we have immediate values of A_3 . B_3 because are i/ps ~~and~~ but ~~C_3~~ I/P C_3 will wait to get value until C_2 is available. ~~to the adder~~. from the previous stage. |||ly C_2 . has to wait for C_1 . & so on. down to C_0.

* Thus only after the carry propagates & ripples through all stages . will the last o/p S_3 & C_4 . settle to their final correct value.

* For this resion the ckt of FA can be redrawn as



* Here I/p C_i to the O/P C_{i+1} propagates through an AND gate & an OR gate.

* two gate levels.
  for 4-FA in adder. O/P C_4 would have
  $2 \times 4 = 8$ gate levels. from C_0 to C_4.

* The carry propagation time is an important characteristic of the adder, because it limits the speed with which two numbers are added.

* An obvious soln for reducing the carry pd.T is to employ faster gates with reduced delays.

* Another s/n is to increasing the ckt complexity to reduce carry delay.

* The most widely used technique ~~om~~ is carry lookahead logic.

Scanned with OKEN Scanner

$P_i = A_i \oplus B_i$

$G_i = A_i B_i$.

o/p S & C

$S_i = P_i \oplus C_i$

$C_{i+1} = G_i + P_i C_i$

G_i — carry generates.

P_i — is called carry propagate.

* Let us write carry o/p ~~&~~ for each stage.

$C_0 =$ I/p carry

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0)$

$= G_1 + P_1 G_0 + P_1 P_0 C_0$.

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$

$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$.

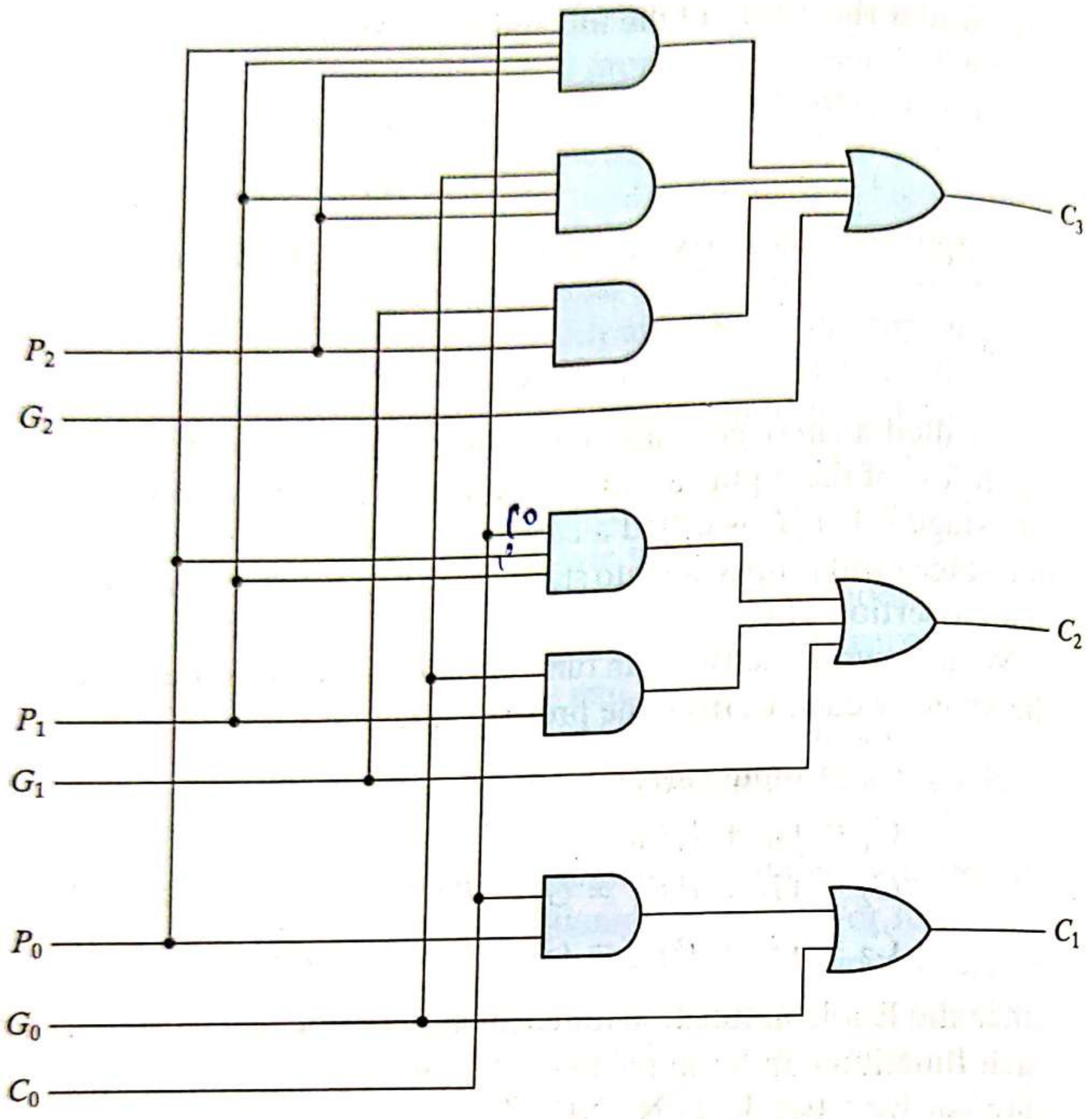


FIGURE 4.11
Logic diagram of carry lookahead generator

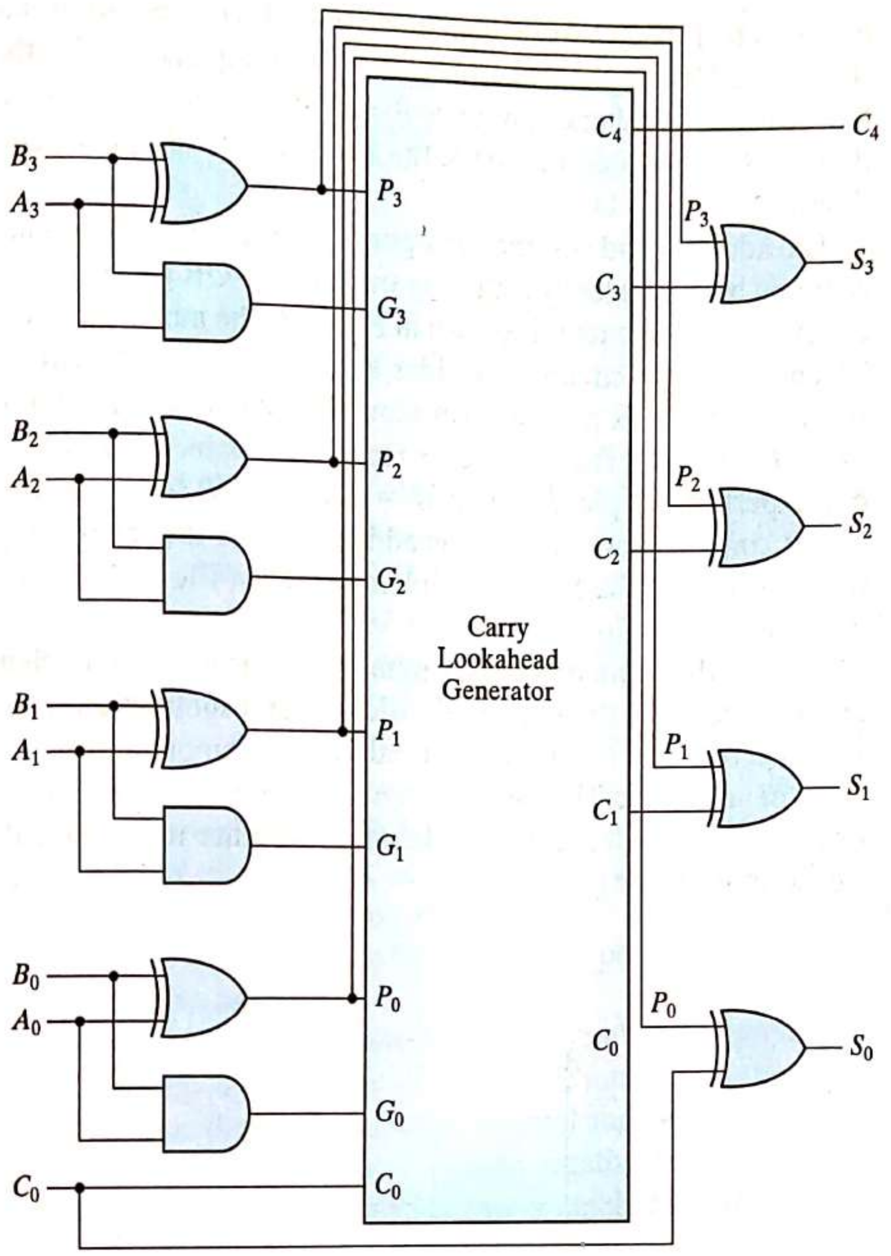


FIGURE 4.12
Four-bit adder with carry lookahead

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements, as discussed in Section 1.5. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry of a full adder.

The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when subtraction is performed. The operation thus performed becomes A, plus the 1's complement of B, plus 1. This is equal to A plus the 2's complement of B. For unsigned numbers, that gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$, provided that there is no overflow. (See Section 1.6.)

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder–subtractor circuit is shown in Fig. 4.13. The mode input M controls the operation. When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B. (The exclusive-OR with output V is for detecting an overflow.)

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as are unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

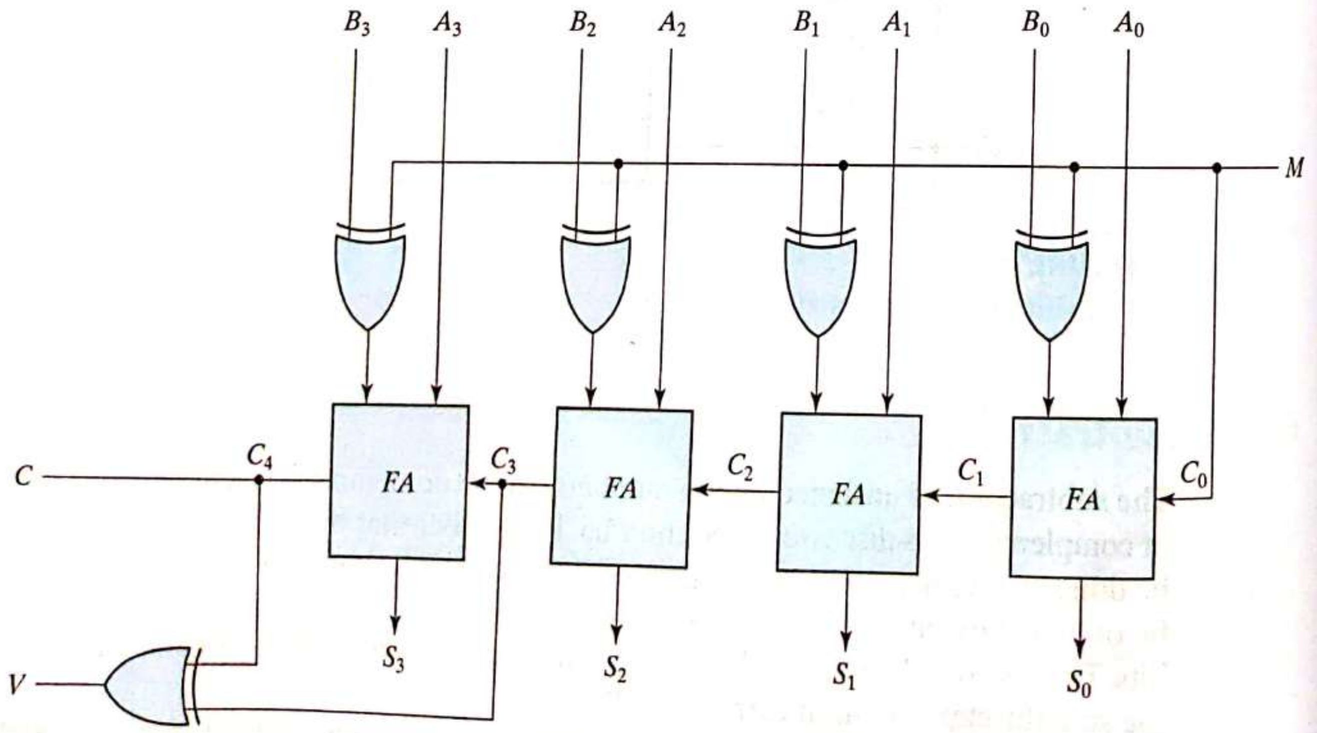


FIGURE 4.13
Four-bit adder–subtractor (with overflow detection)

Practice Exercise 4.6

Find A−B with A = 1001_2 and B = 0110_2;

Answer: A−B = 1_0011_2

Overflow

When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an *overflow* occurred. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n + 1$ bits cannot be accommodated by an n-bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary −128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for −70 and −80. The two additions in binary are shown next, together with the last two carries:

carries:	0 1	carries:	0 1
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
150	1 0010110	−150	0 1101010

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the eighth bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1. For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder–subtractor circuit with outputs C and V is shown in Fig. 4.13. If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow. If $V = 0$ after an addition or subtraction, then no overflow occurred and the n-bit result is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred. The $(n + 1)$ th bit is the actual sign and has been shifted out of position.

4.9 DECODERS

Discrete quantities of information are represented in digital systems by binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of coded information. A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

The decoders presented here are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output. The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

As an example, consider the three-to-eight-line decoder circuit of Fig. 4.18. The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each

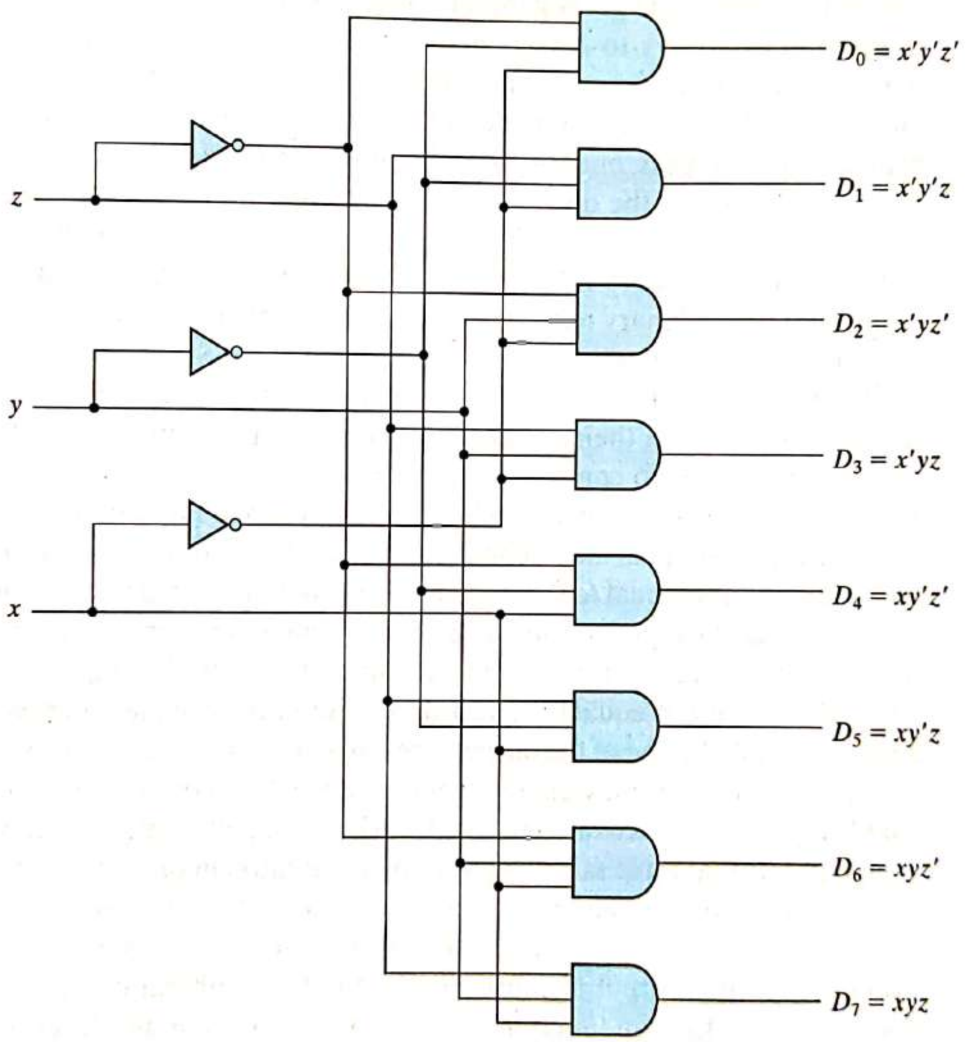


FIGURE 4.18
Three-to-eight-line decoder

Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

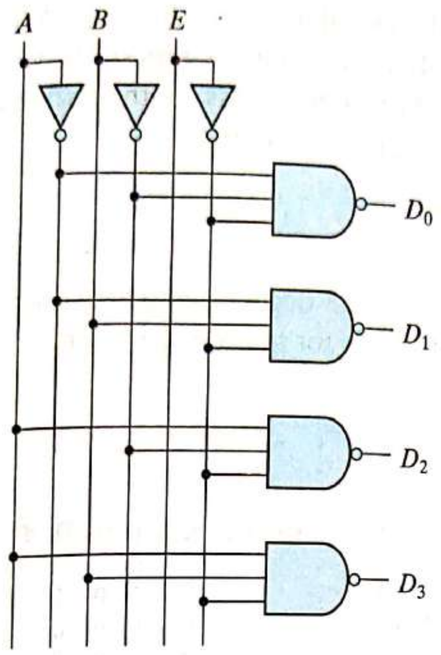
Inputs			Outputs							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

one of the eight AND gates generates one of the minterms. A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for decoding *any* three-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be clarified by the truth table listed in Table 4.6. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more *enable* inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in Fig. 4.19. The circuit operates with complemented outputs and a complement enable input. The outputs of the decoder are enabled when E is equal to 0 (i.e., active-low enable). As indicated by the truth table, only one output can be equal to 0 at any given time; all other outputs are equal to 1. The output whose value is equal to 0 represents the minterm selected by inputs A and B. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal. Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit.

A decoder with enable input can function as a *demultiplexer*—a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines. The decoder of Fig. 4.19 can function as a one-to-four-line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B. This



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

FIGURE 4.19
Two-to-four-line decoder with enable input

feature can be verified from the truth table of the circuit. For example, if the selection lines $AB = 10$, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder demultiplexer*.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure 4.20 shows two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When $w = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w = 1$, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all

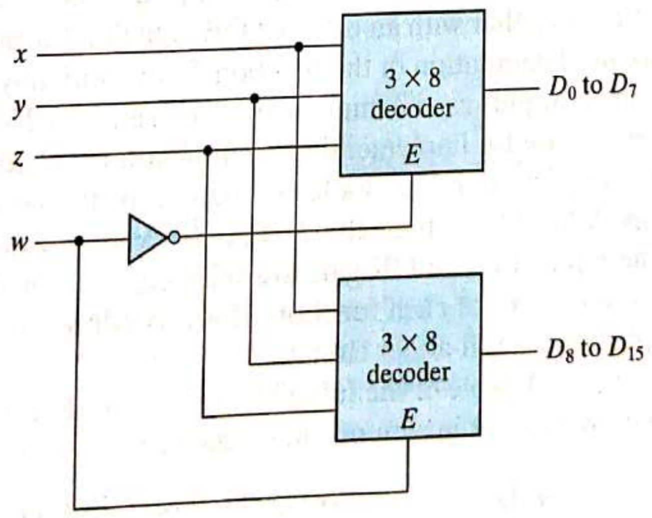


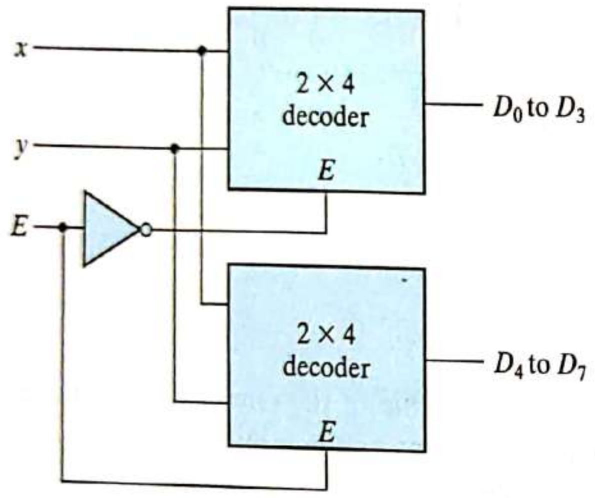
FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

0's. This example demonstrates the usefulness of enable inputs in decoders and other combinational logic components. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.

Practice Exercise 4.8

Draw a logic diagram constructing a 3×8 decoder with active-low enable, using a pair of 2×4 decoders; also draw a truth table for the configuration.

Answer:



E	x	y	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

FIGURE PE4.8

ombinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n-line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean function for the circuit be expressed as a sum of minterms. A decoder is then chosen that generates all the minterms of the input variables. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an example that implements a full-adder circuit.

From the truth table of the full-adder (see Table 4.4), we obtain the functions for the combinational circuit in sum-of-minterms form:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

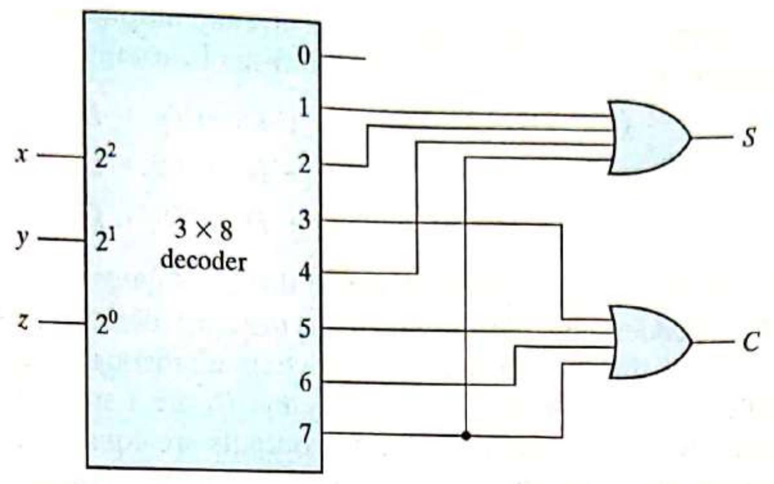


FIGURE 4.21
Implementation of a full adder with a decoder

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder. The implementation is shown in Fig. 4.21. The decoder generates the eight minterms for x, y, and z. The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complemented form F' with $2^n - k$ minterms. If the number of minterms in the function is greater than $2^n/2$, then F' can be expressed with fewer minterms. In such a case, it is advantageous to use a NOR gate to sum the minterms of F'. The output of the NOR gate complements this sum and generates the normal output F. If NAND gates are used for the decoder, as in Fig. 4.19, then the external gates must be NAND gates instead of OR gates. This is because a two-level NAND gate circuit implements a sum-of-minterms function and is equivalent to a two-level AND-OR circuit.

4.10 ENCODERS

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to each input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 4.7. It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates.

The encoder defined in Table 4.7 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

A priority encoder is an encoder circuit that includes the priority function, and handles the possibility that inputs might be in contention. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 4.8. In addition to the two outputs x and y, the circuit has a third output designated by V; this is a *valid* bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input, and V is equal to 0. The other two

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

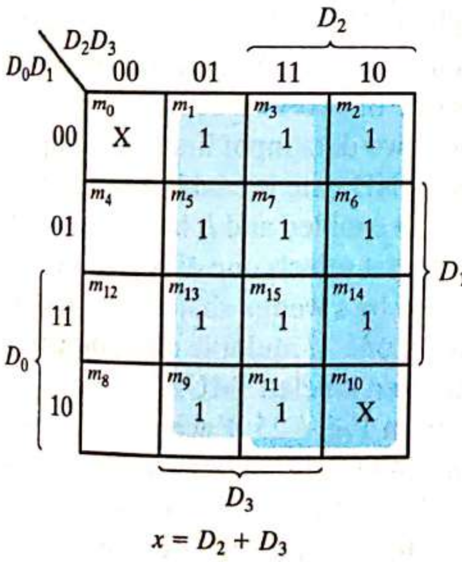
Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

outputs are not inspected when V equals 0, and are specified as don't-care conditions. Note that whereas X's in output columns represent don't-care conditions, the X's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X100 represents the two minterms 0100 and 1100.

According to Table 4.8, the higher the subscript number, the higher the priority of the input is. Input D_3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

The K-maps for simplifying outputs x and y are shown in Fig. 4.22. The minterms for the two functions are derived from Table 4.8. Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the fourth row in the table, with inputs XX10, represents the


D_2D_3
D_0D_1
D_2
D_1
D_0
D_3
00
01
11
10
m_0 X, m_1 1, m_3 1, m_2 1
m_4, m_5 1, m_7 1, m_6 1
m_{12}, m_{13} 1, m_{15} 1, m_{14} 1
m_8, m_9 1, m_{11} 1, m_{10} X
$x = D_2 + D_3$

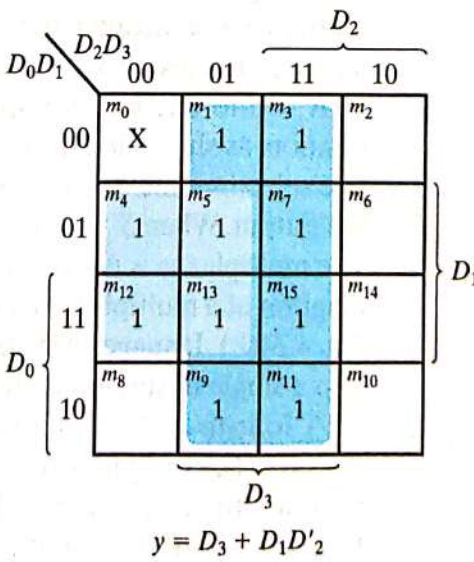

D_2D_3
D_0D_1
D_2
D_1
D_0
D_3
00
01
11
10
m_0 X, m_1 1, m_3 1, m_2
m_4 1, m_5 1, m_7 1, m_6
m_{12} 1, m_{13} 1, m_{15} 1, m_{14}
m_8, m_9 1, m_{11} 1, m_{10}
$y = D_3 + D_1D'_2$

FIGURE 4.22
Maps for a priority encoder

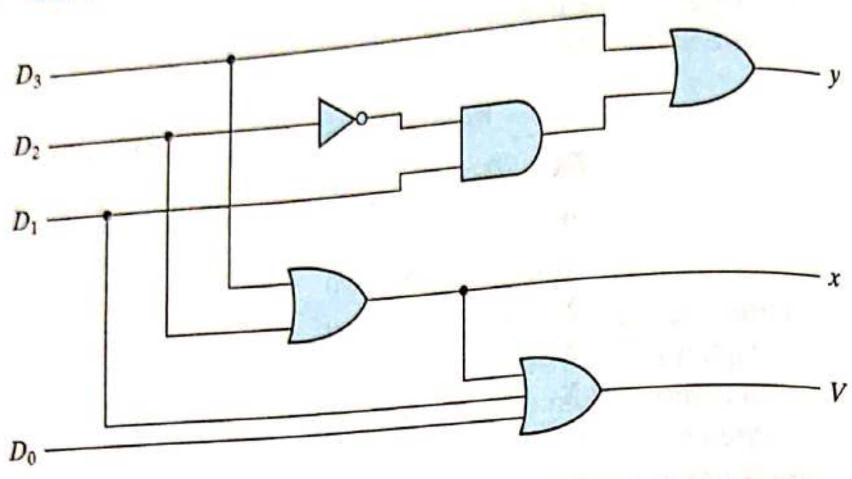


FIGURE 4.23
Four-input priority encoder

four minterms 0010, 0110, 1010, and 1110. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 4.23 according to the following Boolean functions:

$$
\begin{aligned}
x &= D_2 + D_3 \\
y &= D_3 + D_1 D_2' \\
V &= D_0 + D_1 + D_2 + D_3
\end{aligned}
$$

.11 MULTIPLEXERS

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.[6] The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 4.24. The circuit has two data input lines, one output line, and one selection line S. When $S = 0$, the upper AND gate is enabled and I_0 has a path to the output. When $S = 1$, the lower AND gate is enabled and I_1 has a path to the output. The multiplexer acts like an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. 4.24(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled "MUX" in block diagrams.

A four-to-one-line multiplexer is shown in Fig. 4.25. Each of the four inputs, I_0 through I_3, is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output. The function table lists the

[6] Lines represent signals in circuit drawings. It is common usage to refer to input and output signals of a multiplexer as input and output lines.

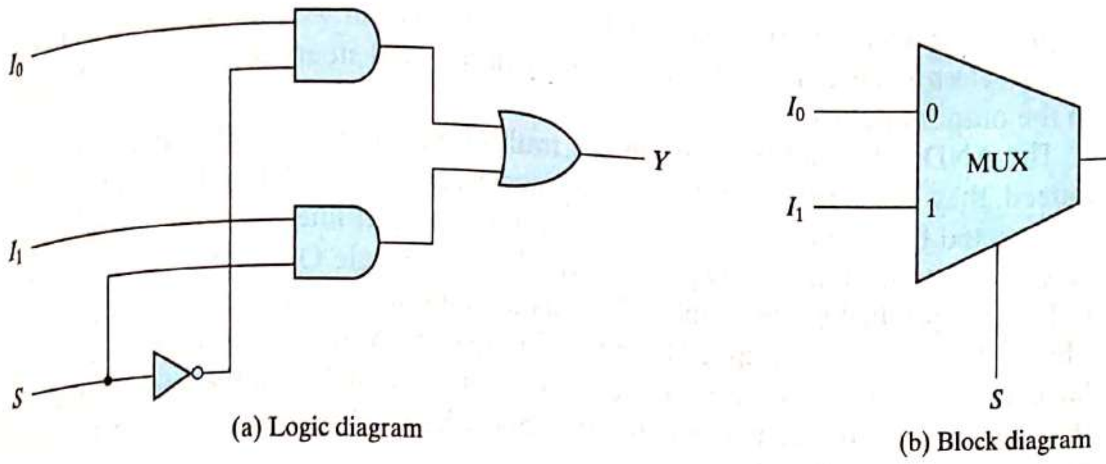
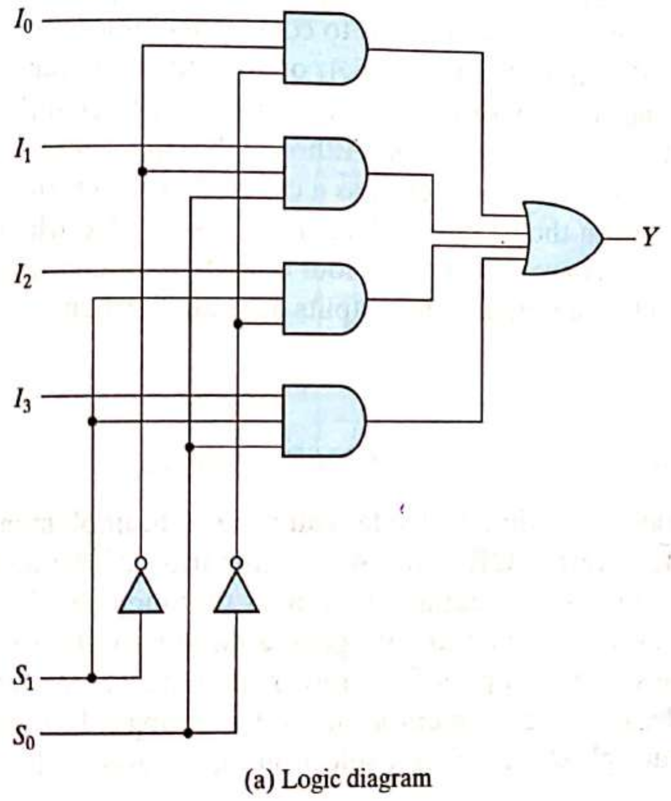


FIGURE 4.24
Two-to-one-line multiplexer



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when $S_1S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The output of the OR gate is now equal to the value of

Boolean Function Implementation with Multiplexers

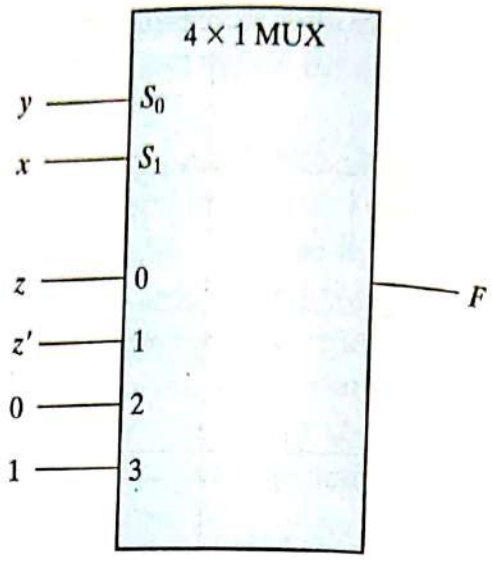
In Section 4.9, it was shown that a decoder can be used to implement Boolean functions by employing external OR gates. An examination of the logic diagram of a multiplexer reveals that it is essentially a decoder that includes the OR gate within the unit. The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs. The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs, one for each minterm.

We will now show a more efficient method for implementing a Boolean function of n variables with a multiplexer that has $n - 1$ selection inputs, instead of n selection inputs. The first $n - 1$ variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable of the function is used for the data inputs. If the single variable is denoted by z, each data input of the multiplexer will be z, z', 1, or 0. To demonstrate this procedure, consider the Boolean function

$$F(x, y, z) = \Sigma(1, 2, 6, 7)$$

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

$xy = 00$, data input 0 has a path to the output, and that makes F equal to z. In a simila fashion, we can determine the required input to data lines 1, 2, and 3 from the value o F when $xy = 01$, 10, and 11, respectively. This particular example shows all four pos-sibilities that can be obtained for the data inputs.

The general procedure for implementing any Boolean function of n variables with a multiplexer with $n - 1$ selection inputs and 2^{n-1} data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first $n - 1$ variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the data inputs in the proper order.

As a second example, consider the implementation of the Boolean function

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

This function is implemented with a multiplexer with three selection inputs as shown in Fig. 4.28. Note that the first variable A must be connected to selection input S_2 so that A, B, and C correspond to selection inputs S_2, S_1, and S_0, respectively. The values for the data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when $ABC = 101$, $F = D$, so the input variable D is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology (e.g., 3 V).

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

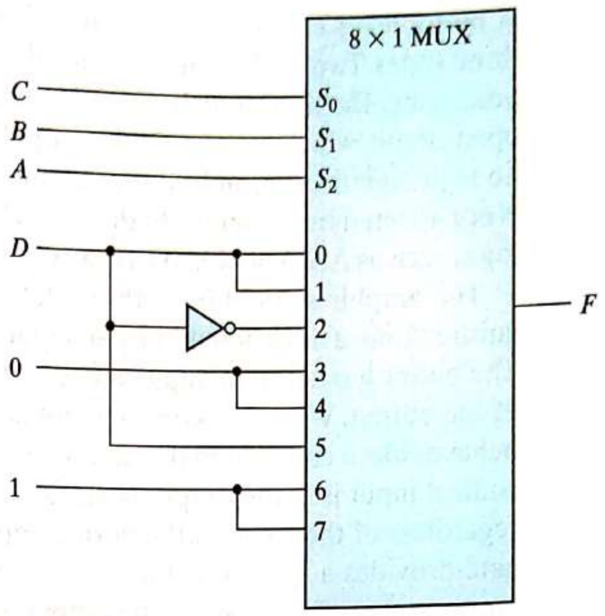


FIGURE 4.28
Implementing a four-input function with a multiplexer

Practice Exercise 4.9

Implement the Boolean function $F(A, B, C) = \Sigma(3, 5, 6, 7)$ with a multiplexer.

Answer:

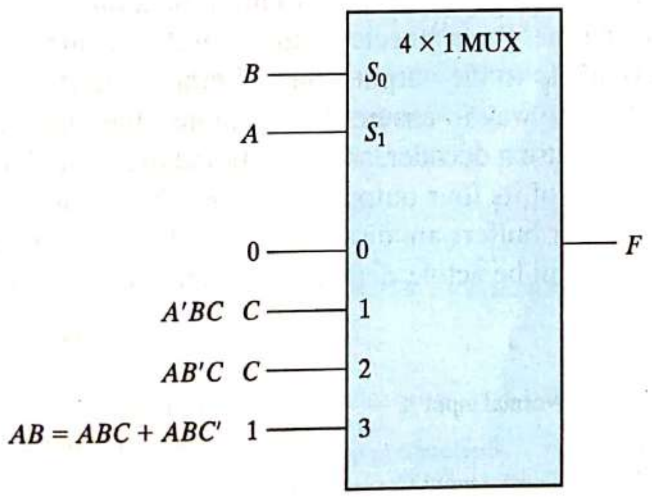


FIGURE PE4.9

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a *high-impedance* state in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and (3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate.

The graphic symbol for a three-state buffer gate is shown in Fig. 4.29. It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30. Figure 4.30(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the selected input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A. When the select input is 1, the lower buffer is enabled and Y is equal to B.

The construction of a four-to-one-line multiplexer is shown in Fig. 4.30(b). The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I_0 through I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three-state buffer has access to the output while all other buffers are maintained in a high-impedance state. One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of

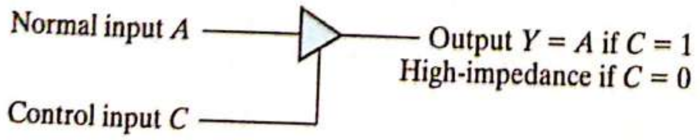


FIGURE 4.29
Graphic symbol for a three-state buffer

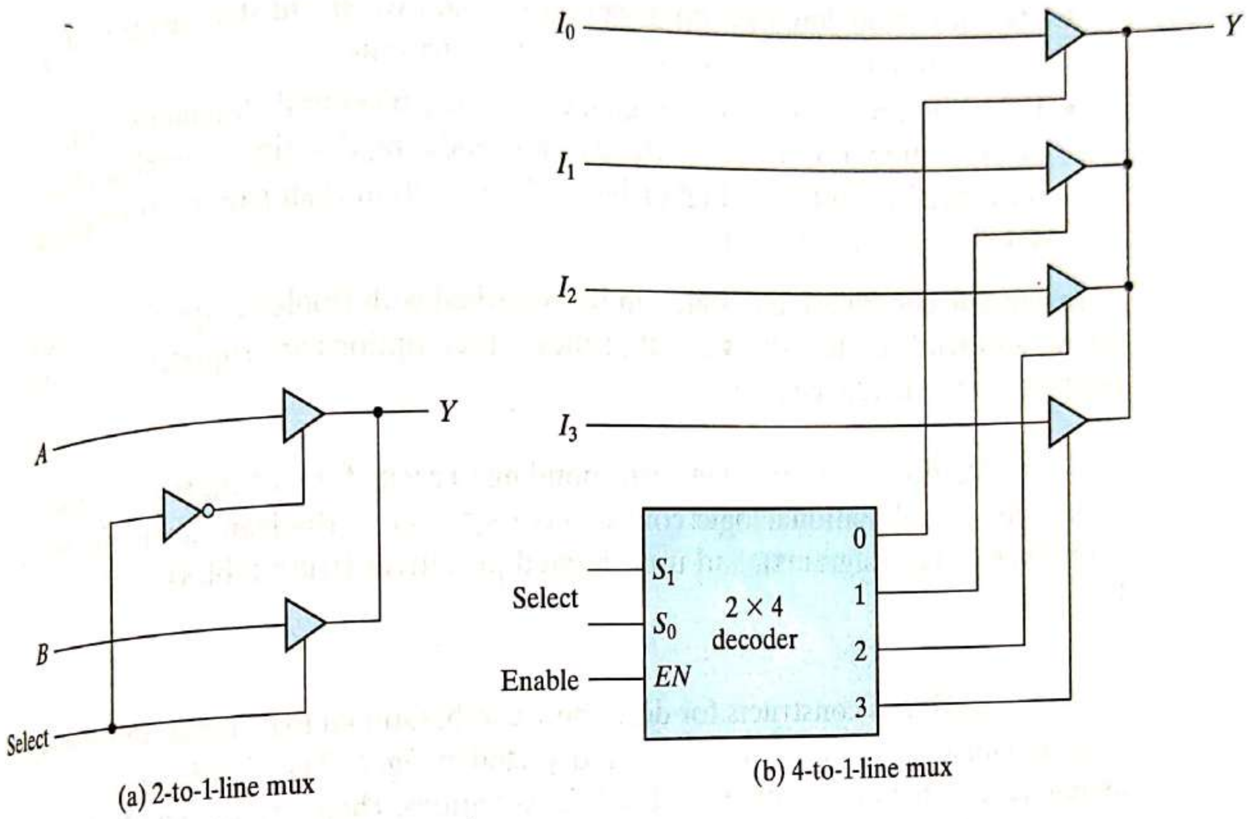


FIGURE 4.30
Multiplexers with three-state gates

the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

Basic features of Verilog and VHDL were introduced in Chapter 3. This section introduces additional features of those languages, presents more elaborate examples, and compares alternative descriptions of combinational circuits.[7]

Verilog and VHDL support three common styles of modeling combinational circuits:

- Gate-level modeling, also called *structural* modeling, instantiates and interconnects basic logic circuits to form a more complex circuit having a desired functionality. Gate-level modeling describes a circuit by specifying its gates and how they are connected with each other.[8]

[7]Sequential circuits and their models are presented in Chapter 5.

[8]Verilog also supports switch-level modeling for directly representing MOS transistor circuits. This style is sometimes used in modeling and simulation, but not in synthesis. We will not use switch-level modeling in this text, but we provide a brief introduction in Appendix A.3. For additional information see the Verilog language reference manual.

Table 4.10
Some Verilog Operators

Symbol	Operation	Symbol	Operation
+	binary addition		
−	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
==	equality		
>	greater than		
<	less than		
{ }	concatenation		
? :	conditional		

HDL Example 4.4 (Dataflow: Two-to-Four Line Decoder)

Verilog

```
// Dataflow description of two-to-four-line decoder
// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (   // Verilog 2001, 2005 syntax
  output  [0: 3]       D,
  input                A, B,
                       enable
 );
  assign  D[0] = !((!A) && (!B) && (!enable)),
   D[1] = !((!A) && B && (!enable)),
   D[2] = ((A) && (! B) && (!enable)),
   D[3] = !(A && B && (!enable));
 endmodule
```

HDL Example 4.5 (Dataflow: Four-Bit Adder)

Verilog

```
// Dataflow description of four-bit adder
// Verilog 2001, 2005 module port syntax
module binary_adder (
  output          C_out,
  output [3: 0] Sum,
  input [3: 0]   A, B,
  input           C_in
);
  assign {C_out, Sum} = A + B + C_in    // Continuous assignment statement
endmodule
```

HDL Example 4.7 (Dataflow: Two-to-One Multiplexer)

Verilog

```
// Dataflow description of two-to-one-line multiplexer
module mux_2x1_df (m_out, A, B, select);
output        m_out;
input         A, B;
input         select;
 assign m_out = (select)? A : B;      // Conditional operator
endmodule
```

[22]The conditional operator is a ternary operator, requiring three operands.

HDL Example 4.8 (Behavioral: Alternative Two-to-Four Line Decoder)

Verilog

```
module decoder_2x4_df_beh (   // Verilog 2001, 2005 syntax
 output  [0: 3]  D,
 input           A, B,
                 enable
);
always @ (A, B, enable) begin
 D[0] <= !((!A) && (!B) && (!enable)),
 D[1] <= !((!A) && B && (!enable)),
 D[2] <= !(A && (!B) && (!enable)),
 D[3] <= !(A && B && (!enable));
end;
endmodule
```

HDL Example 4.9 (Behavioral: Two-to-One Line Multiplexer)

Verilog (Procedural Statement)

```
// Behavioral description of two-to-one-line multiplexer
module mux_2x1_beh (m_out, A, B, select);
  output     m_out;
  input      A, B, select;
  reg        m_out;

  always @ (A or B or select)      // Alternative: always @ (A, B, select)
  if (select == 1) m_out = A;
  else m_out = B;
endmodule
```

Verilog

```
// Behavioral description of four-to-one line multiplexer
// Verilog 2001, 2005 port syntax
module mux_4x1_beh
(output reg  m_out,
  input  in_0, in_1, in_2, in_3,
  input [1: 0]  select
);
always @ (in_0, in_1, in_2, in_3, select)     // Verilog 2001, 2005, SV syntax
 case (select)
  2'b00:  m_out <= in_0;
  2'b01:  m_out <= in_1;
  2'b10:  m_out <= in_2;
  2'b11:  m_out <= in_3;
 endcase
endmodule
```

5.1 INTRODUCTION

Hand-held devices, cell phones, navigation receivers, personal computers, digital cameras, personal media players, and virtually all electronic consumer products have the ability to send, receive, store, retrieve, and process information represented in a binary format. The technology enabling and supporting these devices is critically dependent on electronic components that can store information, that is, have memory. This chapter examines the operation and control of these devices and their use in circuits and enables you to better understand what is happening in these devices when you interact with them. The digital circuits considered thus far have been combinational—their output depends only and immediately on their inputs—they have no memory, that is, they do not depend on past values of their inputs. Sequential circuits, however, act as storage elements and have memory. They can store, retain, and then retrieve information when needed at a later time. It is important that you understand the distinction between sequential and combinational circuits.

5.2 SEQUENTIAL CIRCUITS

Figure 5.1 shows a block diagram of a sequential circuit. It consists of a combinational circuit to which memory elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, **a sequential circuit is specified by a time sequence of inputs, outputs, and internal states.** In contrast, the outputs of combinational logic depend on only the present values of the inputs.

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A *synchronous* sequential circuit is a system whose behavior

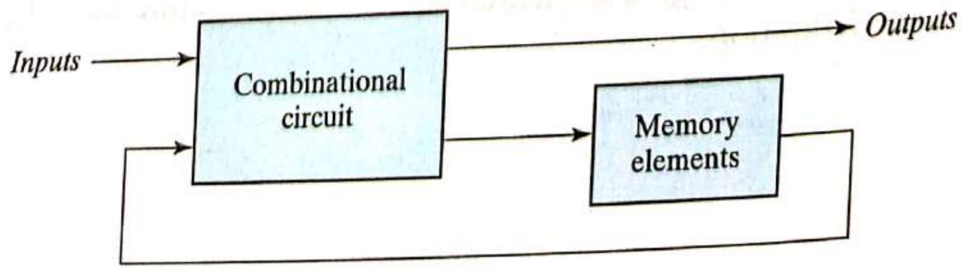
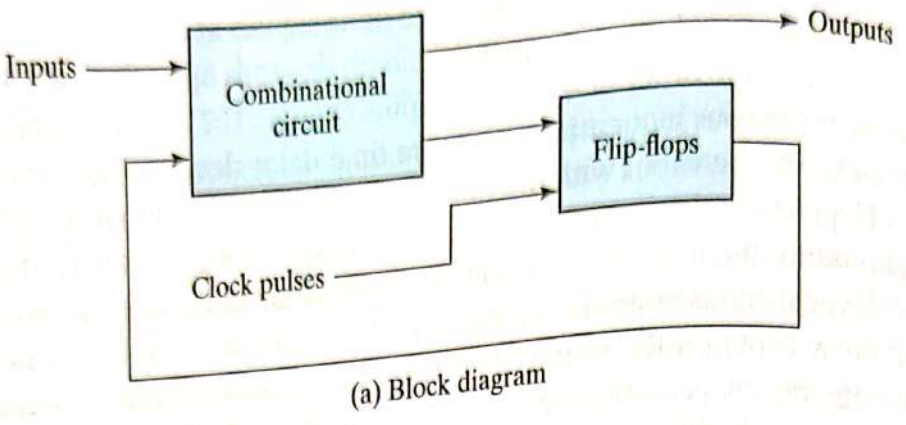


FIGURE 5.1
Block diagram of sequential circuit

can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous* sequential circuit depends upon the input signals at any instant of time *and* the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times. The instability problem imposes many difficulties on the designer, and limits their use. These circuits will not be covered in this text.

A *synchronous* sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic sequence of *clock pulses*. The clock signal is commonly denoted by the identifiers *clock* and *clk*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine *when* computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine *what* changes will take place affecting the storage elements and the outputs. For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits* and are the most frequently encountered type in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems, and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops.* A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. For example, a word of data may be stored as a 64-bit value. The block diagram of a synchronous clocked sequential circuit is shown in Fig. 5.2. The *outputs* are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flip-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig. 5.2, the combinational logic must respond to a change in



(a) Block diagram

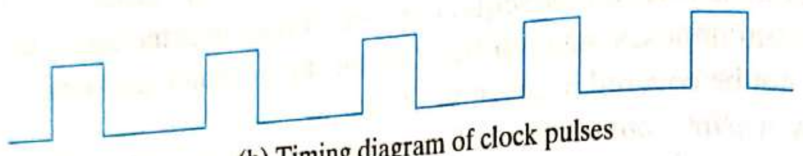

(b) Timing diagram of clock pulses

FIGURE 5.2
Synchronous clocked sequential circuit

the state of the flip-flop in time to be updated before the next pulse arrives. Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. A change in state of the flip-flops is initiated only by a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

Practice Exercise 5.1

Describe the fundamental difference between the output of a combinational circuit and the output of a sequential circuit.

Answer: The output of a combinational circuit depends on only the inputs to the circuit; the output of a sequential circuit depends on the inputs to the circuit and the present state of the storage elements.

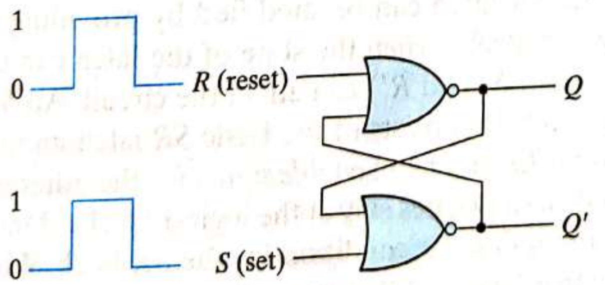
5.3 STORAGE ELEMENTS: LATCHES

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states. The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state. *Storage elements that operate with signal levels (rather than signal transitions) are referred* *to as latches; those controlled by a clock transition are flip-flops.* Latches are said to be *level-sensitive* devices; flip-flops are *edge-sensitive* devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits. Because they are the building blocks of flip-flops, however, we will now consider the fundamental storage mechanism used in latches before considering flip-flops in the next section.

SR Latch

The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set, and *R* for reset. The *SR* latch constructed with two cross-coupled NOR gates is shown in Fig. 5.3. The latch has two useful states. When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $Q' = 1$, it is in the *reset state*. Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state, called a metastable state. Consequently, in practical applications, **setting both inputs to 1 is forbidden**.

Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to (only) the *S* input causes the latch to go to the set state. The *S* input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 5.3(b), two input conditions cause the circuit to be in the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input *S* to bring the circuit to the set state. Removing the active input from *S* leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentarily applying a 1 to the *R* input. The 1 can then be removed from *R*, whereupon the circuit remains in the reset state. Thus, when both inputs *S* and *R* are equal to 0, the latch can be in either the set or the reset

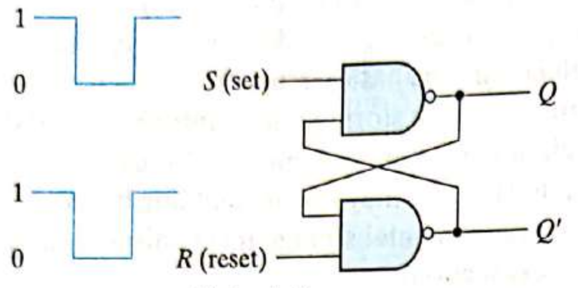


(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 5.3
SR latch with NOR gates



(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table

FIGURE 5.4
SR latch with NAND gates

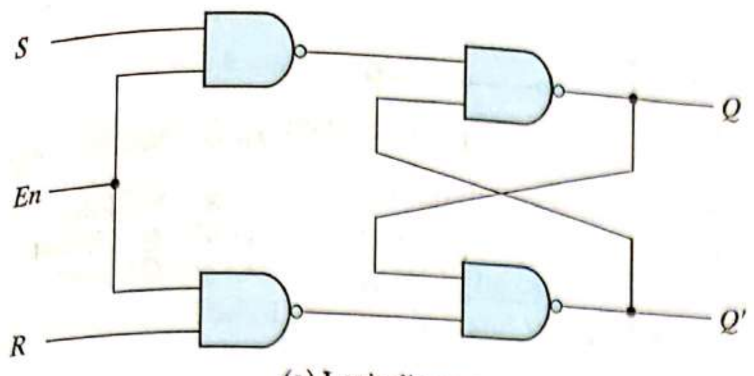
state, depending on which input was most recently a 1. When inputs are applied, the resulting (next) state of the latch depends on the inputs and on the present state of the latch.

If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0. It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.

The *SR* latch with two cross-coupled NAND gates is shown in Fig. 5.4. It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0 to the S input causes output Q to go to 1, putting the latch in the set state. When the S input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the R input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch. The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit.

The operation of the basic *SR* latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed by determining whether S and R (or S' and R') can affect the circuit. An *SR* latch with a control input is shown in Fig. 5.5. It consists of the basic *SR* latch and two additional NAND gates. The control input En acts as an *enable* signal for the other two inputs. The outputs of the two additional NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the *SR* latch. When the enable input goes to 1, information from the S or R input is allowed to affect the latch. The set state is reached with $S = 1, R = 0$, and $En = 1$ (active-high enabled). To change to the reset state, the inputs must be $S = 0, R = 1$, and $En = 1$. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit



(a) Logic diagram

En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

(b) Function table

FIGURE 5.5
SR latch with control input

by applying 0 to *En*, so that the state of the output does not change regardless of the values of *S* and *R*. Moreover, when $En = 1$ and both the *S* and *R* inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.

An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic *SR* latch, which puts it in the undefined state. When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the *S* or *R* input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice. Nevertheless, the *SR* latch is an important circuit because other useful latches and flip-flops are constructed from it.

Practice Exercise 5.2

(a) What input condition puts an *SR* NOR latch into an indeterminate state?

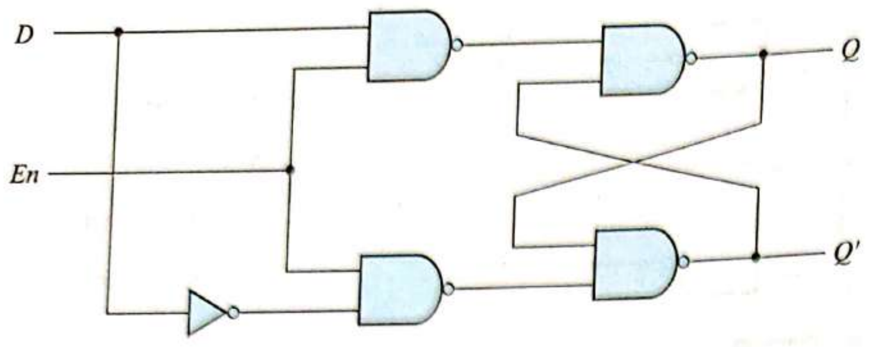
Answer: Both inputs are 1.

(b) What input condition puts an *SR* NAND latch into an indeterminate state?

Answer: Both inputs are 0.

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done in the *D* latch, shown in Fig. 5.6. This latch has only two inputs: *D* (data) and *En* (enable). The *D* input goes directly to the *S* input, and its complement is applied to the *R* input. As long as the enable input is at 0, the cross-coupled *SR* latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of *D*. The *D* input is sampled when $En = 1$. If $D = 1$, the *Q* output goes to 1, placing the circuit in the set state. If $D = 0$, output *Q* goes to 0, placing the circuit in the reset state.



(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(b) Function table

FIGURE 5.6
D latch

The *D* latch receives that designation from its ability to hold *data* in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. *The binary information present at the data input of the* D *latch is transferred to the Q output when the enable input is asserted.* The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input *D* to the output, and for this reason, the circuit is often called a *transparent* latch. When the enable input signal is de-asserted, the binary information that was present at the data input at the time the transition of *enable* occurred is retained (i.e., stored) at the *Q* output until the enable input is asserted again. Note that an inverter could be placed at the enable input. Then, depending on the physical circuit, the external enabling signal will be a value of 0 (active low) or 1 (active high).

The graphic symbols for the various latches are shown in Fig. 5.7. A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the *SR* latch has inputs *S* and *R* indicated inside the block. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the *D* latch has inputs *D* and *En* indicated inside the block.

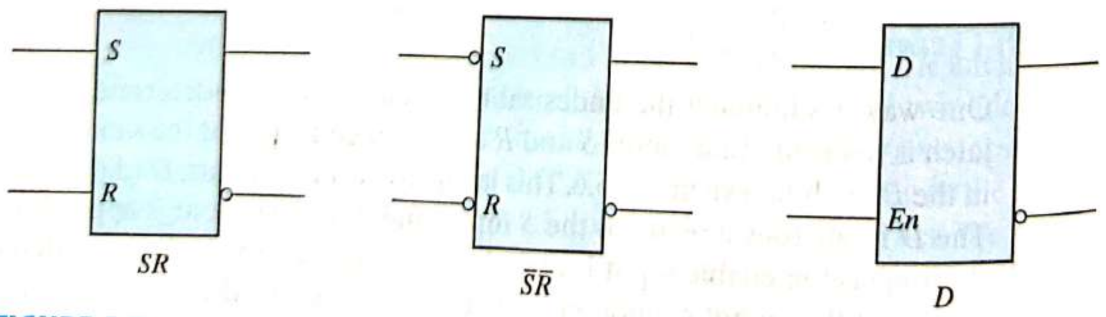


FIGURE 5.7
Graphic symbols for latches

Practice Exercise 5.3

Describe the functionality of a transparent latch.

Answer: A transparent latch has a data input, an enable input, and output. When the enable input is asserted, the output of the latch follows the input to the latch. When the enable input is de-asserted, the output of the latch is held at the value that was present at the moment the enable input was de-asserted.

5.4 STORAGE ELEMENTS: FLIP-FLOPS

A change in the control input of a latch or flip-flop switches its state. This momentary change is called a *trigger*, and the transition it causes is said to trigger the flip-flop. The *D* latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

As seen from the block diagram of Fig. 5.2, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. Consequently, the inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. The problem with the latch is that it responds to a change in the *level* of a clock pulse. As shown in Fig. 5.8(a), a positive level response in the enable input allows changes in the output when the *D* input changes while the clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal *transition*. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. 5.8, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing

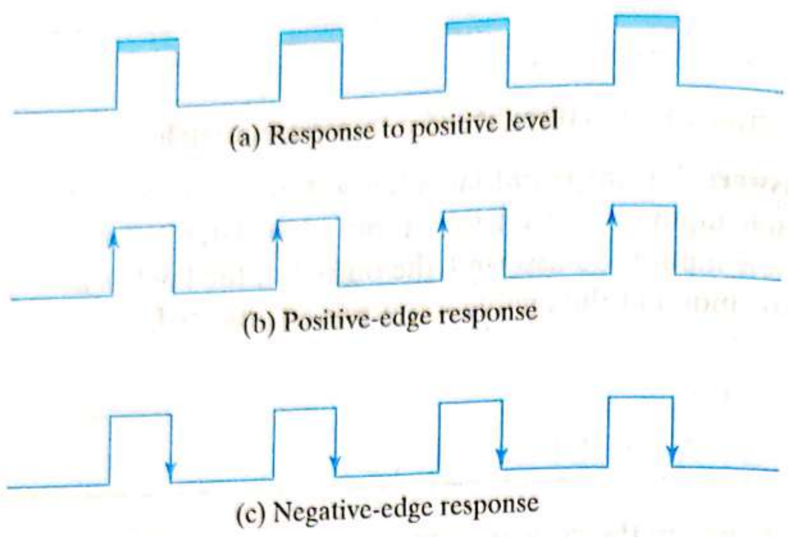

(a) Response to positive level
(b) Positive-edge response
(c) Negative-edge response

FIGURE 5.8
Clock response in latch and flip-flop

signal (clock) and is disabled during the rest of the clock pulse. We will now proceed to show the implementation of both types of flip-flops.

Edge-Triggered *D* Flip-Flop

The construction of a *D* flip-flop with two *D* latches and an inverter is shown in Fig. 5.9. It is often referred to as a master–slave flip-flop. The first latch is called the *master* and the second the *slave*. The circuit samples the *D* input and changes its output *Q* only at the negative edge of the synchronizing or controlling clock (designated as *Clk*). When *Clk* is 0, the output of the inverter is 1. The slave latch is enabled, and its output *Q* is equal to the master output *Y*. The master latch is disabled because $Clk = 0$. When the input (*Clk*) pulse changes to the logic-1 level, the data from the external *D* input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its *enable* input is equal to 0. Any change in the input changes the master output at *Y*, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the *D* input. At the same time, the slave is enabled and the value of *Y* is transferred to the output of the flip-flop at *Q*. Thus, *a change in the output of the flip-flop can be triggered only by and during the transition of the clock from* 1 to 0.

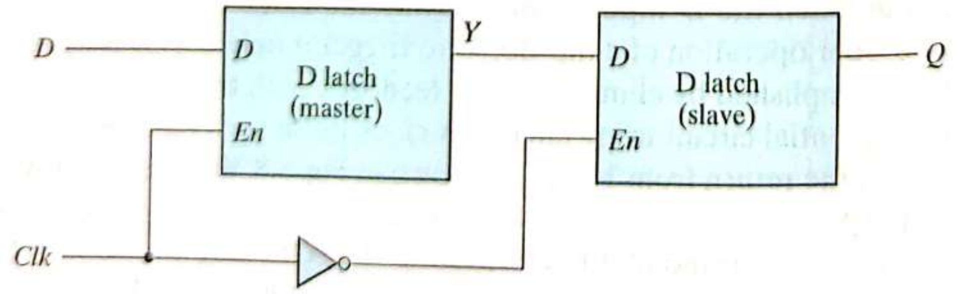


FIGURE 5.9
Master–slave *D* flip-flop

The behavior of the master–slave flip-flop just described dictates that (1) the output may change only once, (2) a change in the output is triggered by the negative edge of the clock, and (3) the change may occur only during the clock's negative level. The value that is produced at the output of the flip-flop is the value that was *stored in the master stage immediately before the negative edge occurred*. It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the *Clk* terminal and the junction between the other inverter and input *En* of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal.

Another construction of an edge-triggered *D* flip-flop uses three *SR* latches as shown in Fig. 5.10. Two latches respond to the external *D* (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop. The *S* and *R* inputs of the output latch are maintained at the logic-1 level when $Clk = 0$. This causes the output to remain in its present state. Input *D* may be equal to 0 or 1. If $D = 0$ when *Clk* becomes 1, *R* changes to 0. This causes the flip-flop to go to the reset state, making $Q = 0$. If there is a change in the *D* input while $Clk = 1$, terminal *R* remains at 0 because *Q* is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, *R* goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if $D = 1$ when *Clk* goes from 0 to 1, *S* changes to 0. This causes the circuit to go to the set state, making $Q = 1$. Any change in *D* while $Clk = 1$ does not affect the output.

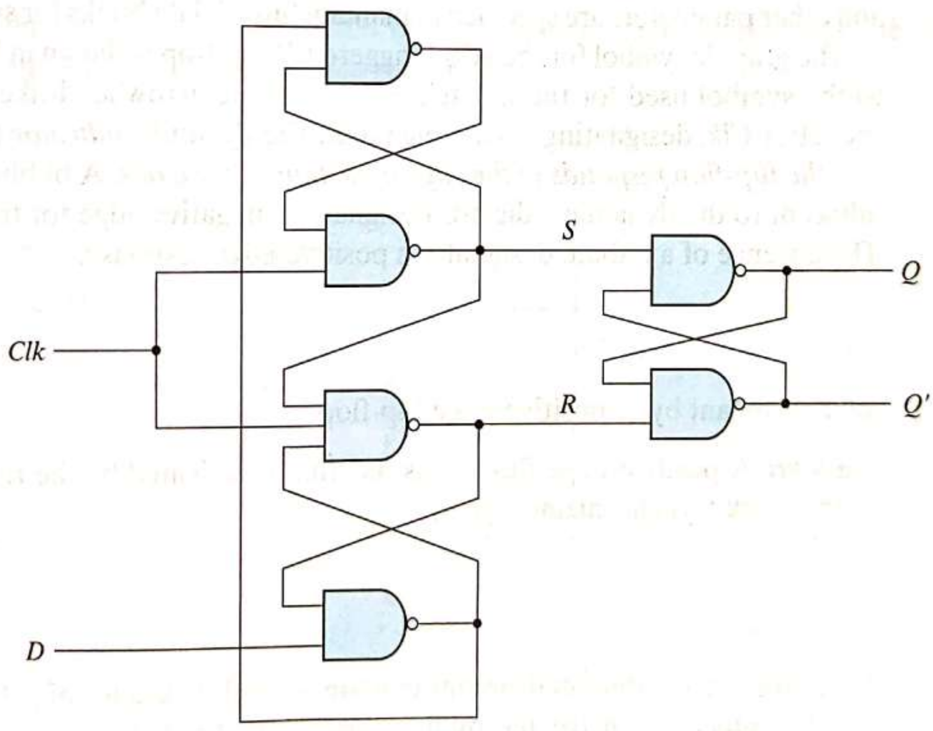


FIGURE 5.10
D-type positive-edge-triggered flip-flop

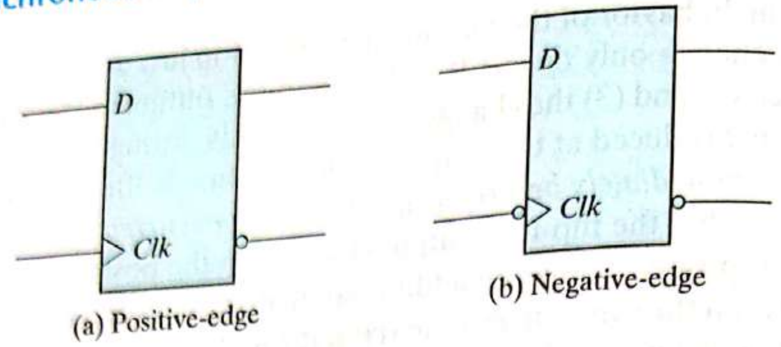


FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

In sum, *when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of* D *is transferred to* Q. A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in *D* when *Clk* is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the *setup time* during which the *D* input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the *hold time* during which the *D* input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state. These and other parameters are specified in manufacturers' data books for specific logic families.

The graphic symbol for the edge-triggered *D* flip-flop is shown in Fig. 5.11. It is similar to the symbol used for the *D* latch, except for the arrowhead-like symbol in front of the letter *Clk*, designating a *dynamic* input. The *dynamic indicator* (>) *denotes the fact that the flip-flop responds to the edge transition of the clock.* A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit. The absence of a bubble designates a positive-edge response.

Practice Exercise 5.4

What is meant by "a positive-edge flip-flop?"

Answer: A positive-edge flip-flop is one that is activated by the rising (positive) edge of the clock (synchronizing signal).

Other Flip-Flops

Very large-scale integrated circuits contain several thousands of gates within one package. Circuits are constructed by interconnecting the various gates to provide a digital system. Each flip-flop is constructed from an interconnection of gates. The most economical and efficient flip-flop constructed in this manner is the edge-triggered

D flip-flop, because it requires the smallest number of gates. Other types of flip-flops can be constructed by using the *D* flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the *JK* and *T* flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the *D* flip-flop can set or reset the output, depending on the value of the *D* input immediately before the clock transition. Synchronized by a clock signal, the *JK* flip-flop has two inputs and performs all three operations. The circuit diagram of a *JK* flip-flop constructed with a *D* flip-flop and gates is shown in Fig. 5.12(a). The *J* input sets the flip-flop to 1, the *K* input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the *D* input:

$$D = JQ' + K'Q$$

When $J = 1$ and $K = 0$, $D = Q' + Q = 1$, so the next clock edge sets the output to 1. When $J = 0$ and $K = 1$, $D = 0$, so the next clock edge resets the output to 0. When both $J = K = 1$ and $D = Q'$ the next clock edge complements the output. When both $J = K = 0$ and $D = Q$, the clock edge leaves the output unchanged. The graphic symbol for the *JK* flip-flop is shown in Fig. 5.12(b). It is similar to the graphic symbol of the *D* flip-flop, except that now the inputs are marked *J* and *K*.

The *T* (toggle) flip-flop is a complementing flip-flop and can be obtained from a *JK* flip-flop when inputs *J* and *K* are tied together. This is shown in Fig. 5.13(a). When $T = 0$ ($J = K = 0$), a clock edge does not change the output. When $T = 1$ ($J = K = 1$), a clock edge complements the output. The complementing flip-flop is useful for designing binary counters.

The *T* flip-flop can be constructed with a *D* flip-flop and an exclusive-OR gate as shown in Fig. 5.13(b). The expression for the *D* input is

$$D = T \oplus Q = TQ' + T'Q$$

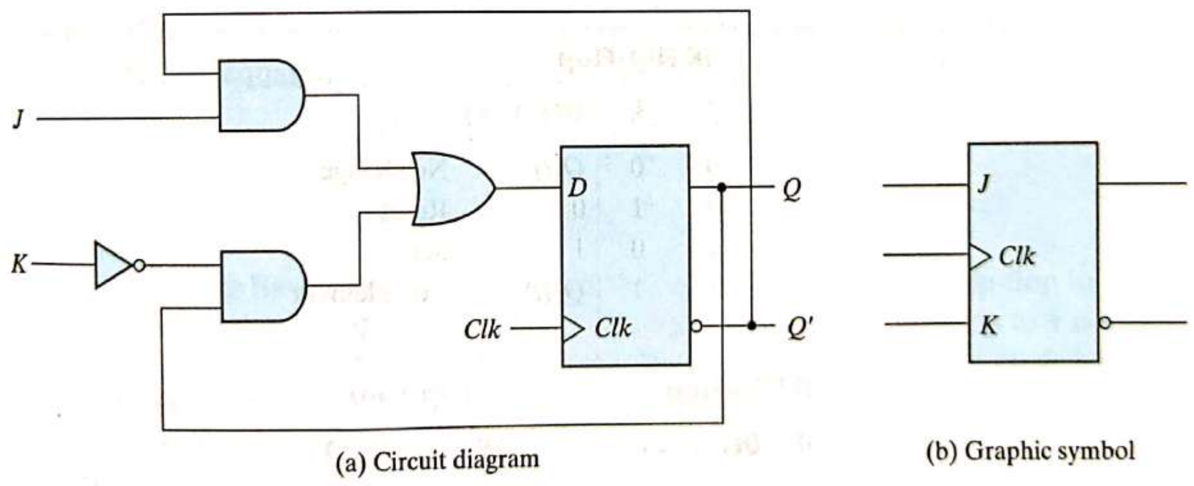


FIGURE 5.12
JK flip-flop

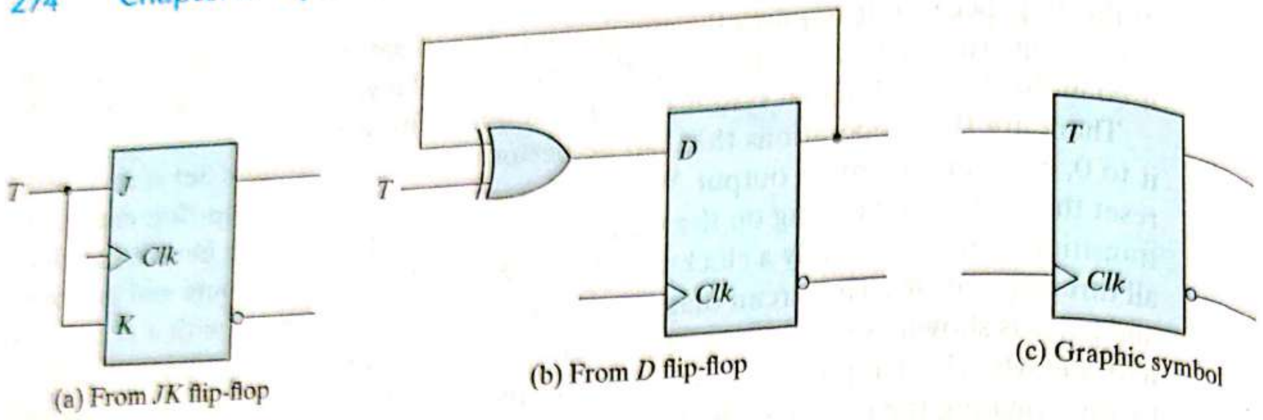


FIGURE 5.13
T flip-flop

When $T = 0, D = Q$ and there is no change in the output. When $T = 1, D = Q'$ and the output complements. The graphic symbol for this flip-flop has a T symbol in the input.

Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 5.1. They define the next state (i.e., the state that results from a clock transition) as a function of the inputs and the present state. $Q(t)$ refers to the present state (i.e., the state present prior to the application of a clock edge). $Q(t + 1)$ is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and $t + 1$. Thus, $Q(t)$ denotes the state of the flip-flop immediately before the clock edge, and $Q(t + 1)$ denotes the state that results from the clock transition.

Table 5.1
Flip-Flop Characteristic Tables

***JK* Flip-Flop**

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

***D* Flip-Flop**

D	Q(t + 1)	
0	0	Reset
1	1	Set

***T* Flip-Flop**

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

The characteristic table for the JK flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as $Q(t + 1) = Q(t)$, indicating that the clock produces no change of state. When $K = 1$ and $J = 0$, the clock resets the flip-flop and $Q(t + 1) = 0$. With $J = 1$ and $K = 0$, the flip-flop sets and $Q(t + 1) = 1$. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as $Q(t + 1) = Q'(t)$.

The next state of a D flip-flop is dependent on only the D input and is independent of the present state. This can be expressed as $Q(t + 1) = D$. It means that the next-state value is equal to the value of D. Note that the D flip-flop does not have a "no-change" condition. Such a condition can be accomplished either by disabling the clock or by operating the clock by having the output of the flip-flop connected into the D input. Either method effectively circulates the output of the flip-flop when the state of the flip-flop must remain unchanged.

The characteristic table of the T flip-flop has only two conditions: When $T = 0$, the clock edge does not change the state; when $T = 1$, the clock edge complements the state of the flip-flop.

Characteristic Equations

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$Q(t + 1) = D$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 5.12. We obtain

$$Q(t + 1) = JQ' + K'Q$$

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 5.13:

$$Q(t + 1) = T \oplus Q = TQ' + T'Q$$

Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called *preset* or *direct set*. The input that clears the flip-flop to 0 is called *clear* or *direct reset*. When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 5.14. The circuit diagram is the same as the one in Fig. 5.10, except for the additional

Basic Structure of Computers & Machine Instructions and Programs

TOPIC: ***Basic Structure of Computers:*** Basic Operational Concepts, Bus Structures, Performance –Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement. ***Machine Instructions and Program:*** Memory Location and Addresses Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions

1. BASIC OPERATIONAL CONCEPT:

The program to be executed is stored in memory. Instructions are accessed from memory to the processor one by one and executed.

STEPS FOR INSTRUCTION EXECUTION

Consider the following instruction

Ex: 1 Add LOCA, R_0

This instruction is in the form of the following instruction format

Opcode Source, Source/ Destination

Where Add is the *operation code,* LOCA is the Memory operand and R_0 is Register operand
This instruction adds the contents of memory location LOCA with the contents of Register R_0 and the result is stored in R_0 Register.
The symbolic representation of this instruction is

$$R_0 \longleftarrow [LOCA] + [R_0]$$

The contents of memory location LOCA and Register R_0 before and after the execution of this instruction is as follows

Before instruction execution	After instruction execution
LOCA = 23H	**LOCA = 23H**
R_0 = 22H	**R_0 = 45H**

The steps for instruction execution are as follows

1. Fetch the instruction from memory into the IR (instruction register in CPU).
2. Decode the instruction 1111000000 10011010
3. Access the first Operand
4. Access the second Operand
5. Perform the operation according to the Opcode (operation code).
6. Store the result into the Destination Memory location or Destination Register.

Ex:2 **Add R_1, R_2, R_3** (3 address instruction format)

This instruction is in the form of the following instruction format

Opcode, Source-1, Source-2, Destination

Where R1 is Source Operand-1, R2 is the Source Operand-2 and R3 is the Destination. This instruction adds the contents of Register R1 with the contents of R2 and the result is placed in R3 Register.

The symbolic representation of this instruction is

$$\mathbf{R3 \longleftarrow [R1] + [R2]}$$

The contents of Registers R1,R2,R3 before and after the execution of this instruction is as follows.

Before instruction execution	After instruction execution
R1 = 24H	**R1 = 24H**
R2 = 34H	**R2 = 34H**
R3 = 38H	**R3 = 58H**

The steps for instruction execution is as follows

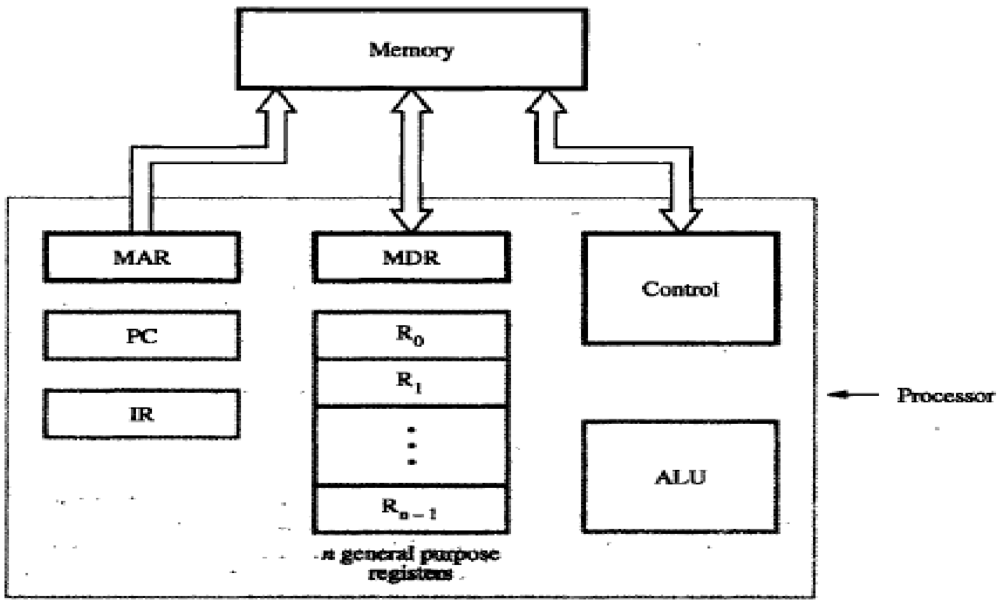
1. Fetch the instruction from memory into the IR.
2. Decode the instruction
3. Access the First Operand R1
4. Access the Second Operand R2
5. Perform the operation according to the Operation Code.
6. Store the result into the Destination Register R3.

CONNECTION BETWEEN MEMORY AND PROCESSOR

The connection between Memory and Processor is as shown in the figure.

The Processor consists of different types of registers.

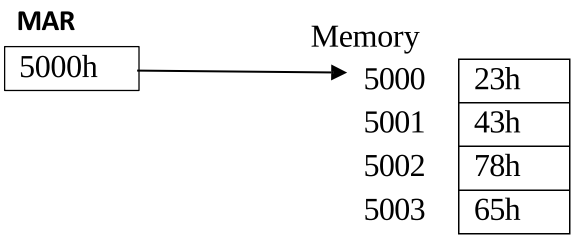
1. MAR (Memory Address Register)
2. MDR (Memory Data Register)
3. Control Unit
4. PC (Program Counter)
5. General Purpose Registers
6. IR (Instruction Register)
7. ALU (Arithmetic and Logic Unit)



The functions of these registers are as follows

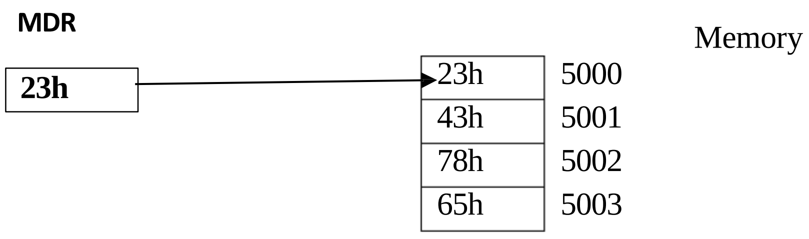
1. **MAR**

- It establishes communication between Memory and Processor
- It stores the address of the Memory Location as shown in the figure.



2. **MDR**

- It also establishes communication between Memory and the Processor.
- It stores the **contents** of the memory location (data or operand), written into or read from memory as shown in the figure.



3. **CONTROL UNIT**

- It controls the data transfer operations between memory and the processor.
- It controls the data transfer operations between I/O and processor.
- It generates control signals for Memory and I/O devices.

4. PC (PROGRAM COUNTER)

- It is a special purpose register used to hold the address of the next instruction to be executed.
- The contents of PC are incremented by 1 or 2 or 4, during the execution of current instruction.
- The contents of PC are incremented by 1 for 8 bit CPU, 2 for 16 bit CPU and for 4 for 32 bit CPU.

4. GENERAL PURPOSE REGISTER / REGISTER ARRAY

The structure of register file is as shown in the figure

R_0
R_1
R_2
.
R_{n-1}

- It consists of set of registers.
- A register is defined as group of flip flops. Each flip flop is designed to store 1 bit of data.
- It is a storage element.
- It is used to store the data temporarily during the execution of the program(eg: result).
- It can be used as a pointer to Memory.
- The Register size depends on the processing speed of the CPU
- EX: Register size = 8 bits for 8 bit CPU

5. IR (INSTRUCTION REGISTER

It holds the instruction to be executed. It notifies the control unit, which generates timing signals that controls various operations in the execution of that instruction.

6. ALU (ARITHMETIC and LOGIC UNIT)

- It performs arithmetic and logical operations on given data.

Steps for fetch the instruction

PC contents are transferred to MAR
Read signal is sent to memory by control unit.
The instruction from memory location is sent to MDR.
The content of MDR is moved to IR.

$$[PC] \rightarrow MAR \xrightarrow[\text{CU (read signal)}]{} Memory \rightarrow MDR \rightarrow IR$$

2. BUS STRUCTURE

Bus is defined as set of parallel wires used for data communication between different parts of computer. Each wire carries 1 bit of data. There are 3 types of buses, namely

1. Address bus
2. Data bus and
3. Control bus1.

1. ***Address bus :***

- It is unidirectional.
- The processor (CPU) sends the address of an I/O device or Memory device by means of this bus.

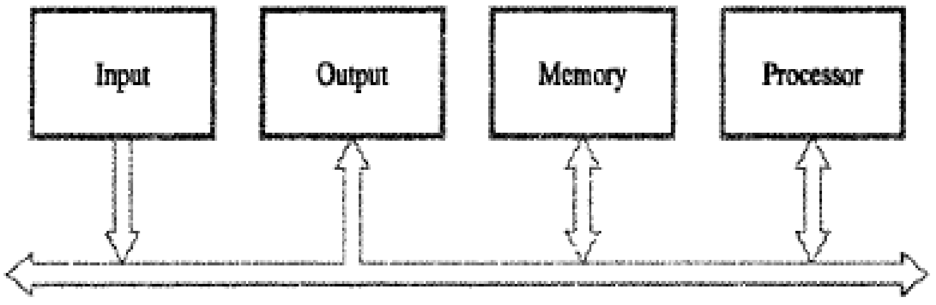
2. Data bus

- It is a bidirectional bus.
- The CPU sends data from Memory to CPU and vice versa as well as from I/O to CPU and vice versa by means of this bus.

3. ***Control bus:***

- This bus carries control signals for Memory and I/O devices. It generates control signals for Memory namely MEMRD and MEMWR and control signals for I/O devices namely IORD and IOWR.

The structure of single bus organization is as shown in the figure.



- The I/O devices, Memory and CPU are connected to this bus is as shown in the figure.
- It establishes communication between two devices, at a time.

Features of Single bus organization are

- Less Expensive
- Flexible to connect I/O devices.
- Poor performance due to single bus.

There is a variation in the devices connected to this bus in terms of speed of operation. Few devices like keyboard, are very slow. Devices like optical disk are faster. Memory and processor are faster, but all these devices uses the same bus. Hence to provide the synchronization between two devices, a buffer register is attached to each device. It holds the data temporarily during the data transfer between two devices.

3. *PERFORMANCE*

Basic performance Equation

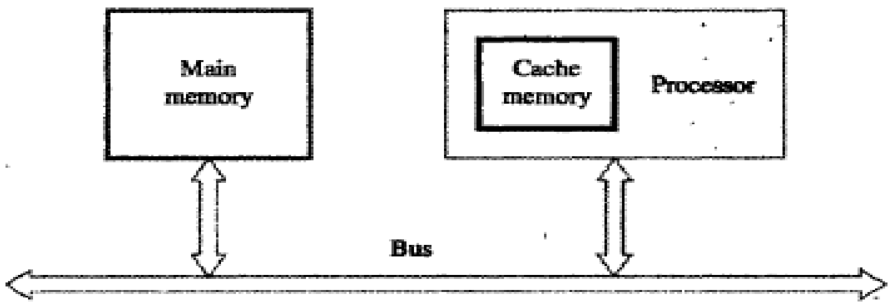
- The performance of a Computer System is based on hardware design of the processor and the instruction set of the processors.
- To obtain high performance of computer system it is necessary to reduce the execution time of the processor.
- Execution time: It is defined as total time required executing one complete program.
- The processing time of a program includes time taken to read inputs, display outputs, system services, execution time etc.
- The performance of the processor is inversely proportional to execution time of the processor.

More performance = Less Execution time.
Less Performance = More Execution time.

The Performance of the Computer System is based on the following factors

1. *Cache Memory*
2. *Processor clock*
3. *Basic Performance Equation*
4. *Instructions*
5. *Compiler*

***CACHE MEMORY*:** It is defined as a *fast access memory* located in between CPU and Memory. It is part of the processor as shown in the fig



The processor needs more time to read the data and instructions from main memory because main memory is away from the processor as shown in the figure. Hence it slowdown the performance of the system.

The processor needs less time to read the data and instructions from Cache Memory because it is part of the processor. Hence it improves the performance of the system.

***PROCESSOR CLOCK*:** The processor circuits are controlled by timing signals called as Clock. It defines constant time intervals and are called as Clock Cycles. To execute one instruction there are 3 basic steps namely

1. Fetch
2. Decode
3. Execute.

The processor uses one clock cycle to perform one operation as shown in the figure

Clock Cycle	→	T1	T2	T3
Instruction	→	Fetch	Decode	Execute

The performance of the processor depends on the length of the clock cycle. To obtain high performance reduce the length of the clock cycle. Let ‘ P ’ be the number of clock cycles generated by the Processor and ‘ R ‘ be the Clock rate .

The Clock rate is inversely proportional to the number of clock cycles.
i.e $R = 1/P$.
Cycles/second is measured in Hertz (Hz). Eg: 500MHz, 1.25GHz.

Two ways to increase the clock rate –

- Improve the IC technology by making the logical circuit work faster, so that the time taken for the basic steps reduces.
- Reduce the clock period, P.

BASIC PERFORMANCE EQUATION

Let ‘ T ‘ be *total time* required to execute the program.
Let ‘N ‘ be the *number of instructions* contained in the program.
Let ‘ S ‘ be the *average number of steps* required to execute one instruction.
Let ‘ R’ be number of clock cycles per second generated by the processor to execute one program.

Processor Execution Time is given by

$$\mathbf{T = N * S / R}$$

This equation is called as Basic Performance Equation.
For the programmer the value of T is important. To obtain high performance it is necessary to reduce the values of N & S and increase the value of R

Performance of a computer can also be measured by using **benchmark** programs.
SPEC (System Performance Evaluation Corporation) is an non-profitable organization, that measures performance of computer using SPEC rating. The organization publishes the application programs and also time taken to execute these programs in standard systems.

$$SPEC = \frac{Running\ time\ of\ reference\ Computer}{Running\ time\ of\ computer\ under\ test}$$

DIFFERENCES MULTIPROCESSOR AND MULTICOMPUTER

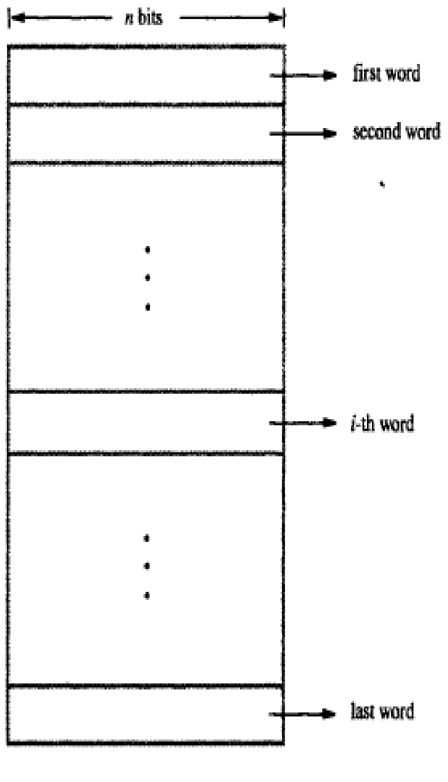
MULTIPROCESSOR	MULTICOMPUTER
1. Interconnection of two or more processors by means of system bus.	Interconnection of two or more computers by means of cables.
2. It uses common memory to hold the data and instructions.	It has its own memory to store data and instructions.
3. Complexity in hardware design.	Not much complexity in hardware design.
4. Difficult to program for multiprocessor system.	Easy to program for multiprocessor system

4. MEMORY LOCATIONS AND ADDRESSES

1. Memory is a storage device. It is used to store character operands, data operands and instructions.
2. It consists of number of semiconductor cells and each cell holds 1 bit of information. A group of 8 bits is called as byte and a group of 16 or 32 or 64 bits is called as word.

World length = 16 for 16 bit CPU and World length = 32 for 32 bit CPU. Word length is defined as number of bits in a word.

- Memory is organized in terms of bytes or words.
- The organization of memory for 32 bit processor is as shown in the fig.



The contents of memory location can be accessed for read and write operation. The memory is accessed either by specifying address of the memory location or by name of the memory location.

- **Address space :** It is defined as number of bytes accessible to CPU and it depends on the number of address lines.

5. BYTE ADDRESSABILITY

Each byte of the memory are addressed, this addressing used in most computers are called byte addressability. Hence Byte Addressability is the process of assignment of address to successive bytes of the memory. The successive bytes have the addresses 1, 2, 3, 4…………2^n-1. The memory is accessed in words.

In a 32 bit machine, each word is 32 bit and the successive addresses are 0,4,8,12,… and so on.

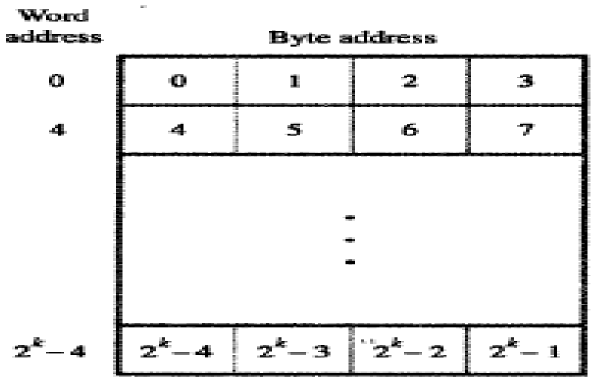
Address	32 – bit word			
0000	0th byte	1st byte	2nd byte	3rd byte
0004	4th byte	5th byte	6th byte	7th byte
0008	8th byte	9th byte	10th byte	11th byte
0012	12th byte	13th byte	14th byte	15th byte
…..	…..	…..	…..	…..
n-3	n-3th byte	n-2th byte	n-1th byte	nth byte

BIG ENDIAN and LITTLE ENDIAN ASSIGNMENT

Two ways in which byte addresses can be assigned in a word.
Or
Two ways in which a word is stored in memory.

1. Big endian
2. Little endian

BIG ENDIAN ASSIGNMENT



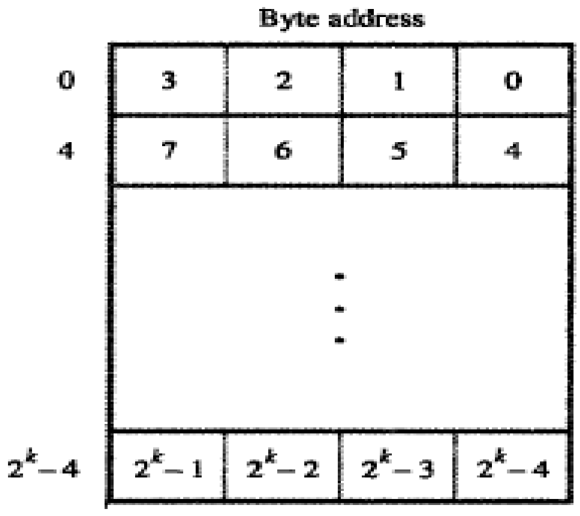
In this technique lower byte of data is assigned to higher address of the memory and higher byte of data is assigned to lower address of the memory.

The structure of memory to represent 32 bit number for big endian assignment is as shown in the above figure.

LITTLE ENDIAN ASSIGNMENT

In this technique lower byte of data is assigned to lower address of the memory and higher byte of data is assigned to higher address of the memory.

The structure of memory to represent 32 bit number for little endian assignment is as shown in the fig.



Eg – store a word “JOHNSENA” in memory starting from word 1000, using Big Endian and Little endian.

Bigendian -

1000	J **1000**	O **1001**	H **1002**	N **1003**
1004	S **1004**	E **1005**	N **1006**	A **1007**

Little endian -

1000	N **1000**	H **1001**	O **1002**	J **1003**
1004	A **1004**	N **1005**	E **1006**	S **1007**

WORD ALLIGNMENT

Word is the group of bytes in memory. Number of bits in a word is the word length.
Eg – 32-bit word length, 64-bit word length etc.

The word locations of memory are aligned, if they begin with the address, which is multiple of number of bytes in a word.

The structure of memory for 16 bit CPU, 32 bit CPU and 64 bit CPU are as shown in the figures 1,2 and 3 respectively

For 16 bit CPU

4000	**34H**
4002	**65H**
4004	**86H**
4006	**93H**
4008	**45H**

(Here, no. of bytes of a word is 2, and the address of word is in multiples of 2)

For 32 bit CPU

4000	**34H**
4004	**65H**
4008	**86H**
4012	**93H**
4016	**45H**

(Here, no. of bytes of a word is 4, and the address of word is in multiples of 4)

For 64 bit CPU

4000	**34H**
4008	**65H**
4016	**86H**
4024	**93H**
4032	**45H**

(Here, no. of bytes of a word is 8, and the address of word is in multiples of 8)

ACCESSING CHARACTERS AND NUMBERS

The character occupies 1 byte of memory and hence byte address for memory.
The numbers occupies 2 bytes of memory and hence word address for numbers.

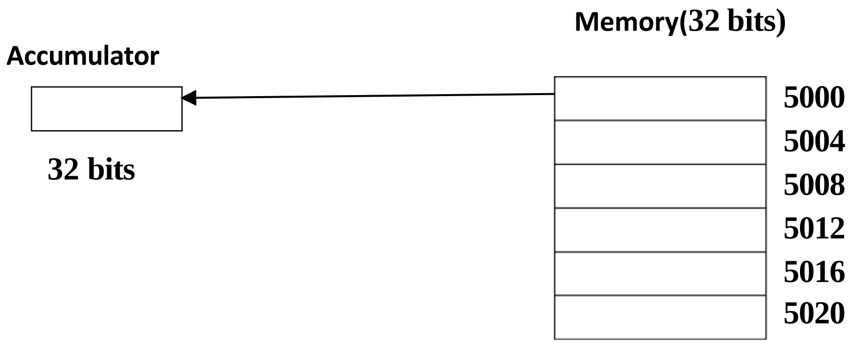
6. MEMORY OPERATION

Both program instructions and operands are in memory.
To execute an instruction, each instruction has to be read from memory and after execution the results must be written to memory.

There are two types of memory operations namely 1. Memory read and 2. Memory write
Memory read operation [Load/ Read / Fetch]
Memory write operation [Store/ write]

1. MEMORY READ OPERATION:

- ✓ It is the process of transferring of 1 word of data from memory into Accumulator (GPR).
- ✓ It is also called as Memory fetch operation.
- ✓ The Memory read operation can be implemented by means of LOAD instruction.
- ✓ The LOAD instruction transfers 1 word of data (1 word = 32 bits) from Memory into the Accumulator as shown in the fig.



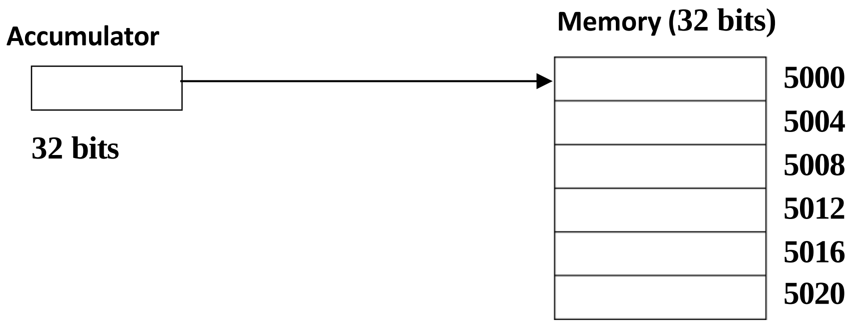
Steps for Memory Read Operation

(1) The processor loads MAR (Memory Address Register) with the address of the memory location.
(2) The Control unit of processor issues memory read control signal to enable the memory component for read operation.
(3) The processor reads the data from memory into the MDR (Memory Data Register) by means of bi-directional data bus.

[MAR] → Memory → MDR

2. MEMORY WRITE OPERATION

- It is the process of transferring the 1 word of data from Accumulator into the Memory.
- The Memory write operation can be implemented by means of STORE instruction.
  The STORE instruction transfers 1 word of data from Accumulator into the Memory location as shown in the fig.



Steps for Memory Write Operation

- The processor loads MAR with the address of the Memory location.
- The processor loads MDR with the data to be stored in Memory location.
- The Control Unit issues the Memory Write control signal.
- The processor transfers 1 word of data from MDR to Memory location by means of bi-directional data bus.

7. COMPUTER OPERATIONS (OR) INSTRUCTIONS AND INSTRUCTION EXECUTION

The Computer is designed to perform 4 types of operations, namely

- Data transfer operations
- ALU Operations
- Program sequencing and control.
- I/O Operations.

1. Data Transfer Operations

a) Data transfer between two registers.

Format: Opcode Source1 , Destination

The processor uses MOV instruction to perform data transfer operation between two registers
The mathematical representation of this instruction is $R1 \rightarrow R2$.

Ex : MOV R_1 , R_2 : R1 and R2 are the registers.

Where MOV is the operation code, R1 is the source operand and R2 is the destination operand.
This instruction transfers the contents of R1 to R2.

EX: Before the execution of MOV R1,R2, the contents of R1 and R2 are as follows

R1 = 34h and R2 = 65h

After the execution of MOV R1, R2, the contents of R1 and R2 are as follows

R1 = 34H and R2 = 34H

b) Data transfer from memory to register

The processor uses **LOAD** instruction to perform data transfer operation from memory to register. The mathematical representation of this instruction is
ACC ←[LOCA]. Where ACC is the Accumulator.

Format : opcode operand

Ex: LOAD LOCA

For this instruction Memory Location is the source and Accumulator is the destination.

c) Data transfer from Accumulator register to memory

The processor uses **STORE** instruction to perform data transfer operation from Accumulator register to memory location. The mathematical representation of this instruction is

LOCA ←[ACC]. Where, ACC is the Accumulator.

Format: opcode operand

Ex: STORE LOCA

For this instruction accumulator is the source and memory location is the destination.

2. ALU Operations

The instructions are designed to perform arithmetic operations such as Addition, Subtraction, Multiplication and Division as well as logical operations such as AND, OR and NOT operations.

Ex1: ADD R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] + [R_1]$; Adds the content of R_0 with the content of R_1 and result is placed in R_1.

Ex2: SUB R_0, R_1

The mathematical representation of this instruction is as follows:

$R_1 \leftarrow [R_0] - [R_1]$; Subtracts the content of R_0 from the content of R_1 and result is placed in R_1.

EX3: AND R_0, R_1 ; It Logically multiplies the content of R_0 with the content of R_1 and result is stored in R_1. ($R_1 = R_0$ AND R1)

3. **I/O Operations:** The instructions are designed to perform INPUT and OUTPUT operations. The processor uses MOV instruction to perform I/O operations.

   The input Device consists of one temporary register called as DATAIN register and output register consists of one temporary register called as DATAOUT register.

   a) Input Operation: It is a process of transferring one WORD of data from DATA IN register to processor register.

   Ex: MOV DATAIN, R0

   The mathematical representation of this instruction is as follows,

   $R_0 \leftarrow [\text{DATAIN}]$

   b) Output Operation: It is a process of transferring one WORD of data from processor register to DATAOUT register.

   Ex: MOV R_0, DATAOUT

   The mathematical representation of this instruction is as follows,

   $[R_0] \rightarrow \text{DATAOUT}$

REGISTER TRANSFER NOTATION

There are 3 locations to store the operands during the execution of the program namely
1. Register 2. Memory location 3. I/O Port. Location is the storage space used to store the data.

- The instructions are designed to transfer data from one location to another location.

Eg 1 - Consider the first statement to transfer data from one location to another location

- " Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."
- The mathematical representation of this statement is given by

  $R_0 \leftarrow [\text{AMOUNT}]$

Eg 2 -Consider the second statement to add data between two registers

- "Add the contents of R_0 with the contents of R_1 and result is stored in R_2"
- The mathematical representation of this statement is given by

  $R_2 \leftarrow [R_0] + [R_1]$.

Such a notation is called as "Register Transfer Notation".

It uses two symbols

1. A pair of square brackets [] to indicate the contents of Memory location and
2. $\leftarrow$ to indicate the data transfer operation.

ASSEMBLY LANGUAGE NOTATION

Consider the first statement to transfer data from one location to another location

"Transfer the contents of Memory location whose symbolic name is given by AMOUNT into processor register R_0."

The assembly language notation of this statement is given by

MOV	AMOUNT,	R_0
Opcode	Source	Destination

This instruction transfers 1 word of data from Memory location whose symbolic name is given by AMOUNT into the processor register R_0.

The mathematical representation of this statement is given by

$$R_0 \leftarrow [AMOUNT]$$

Consider the second statement to add data between two registers

"Add the contents of R_0 with the contents of R_1 and result is stored in R_2"

The assembly language notation of this statement is given by

ADD	R_0 ,	R_1,	R_2
Opcode	source1,	Source2,	Destination

This instruction adds the contents of R_0 with the contents of R_1 and result is stored in R_2.

- The mathematical representation of this statement is given by

$$R_2 \leftarrow [R_0] + [R_1].$$

Such a notations are called as "Assembly Language Notations"

BASIC INSTRUCTION TYPES

There are 3 types of basic instructions namely

1. Three address instruction format
2. Two address instruction format
3. One address instruction format

Consider the arithmetic expression Z = A + B, Where A,B,Z are the Memory locations.

Steps for evaluation

1. Access the first memory operand whose symbolic name is given by A.
2. Access the second memory operand whose symbolic name is given by B.
3. Perform the addition operation between two memory operands.
4. Store the result into the 3rd memory location Z.
5. The mathematical representation is $Z \leftarrow [A] + [B]$.

a) Three address instruction format : Its format is as follows

opcode	**Source-1**	**Source-2**	**destination**

Destination ← [source-1] + [source-2]

Ex: ADD A, B, Z

$Z \leftarrow [A] + [B]$

a) Two address instruction format : Its format is as follows

opcode	**Source**	**Source/destination**

Destination ← [source] + [destination]
The sequence of two address m/c instructions to evaluate the arithmetic expression
Z ← A + B are as follows

MOV A, R_0
MOV B, R_1
ADD R_0, R_1
MOV R_1, Z

b) One address instruction format : Its format is as follows

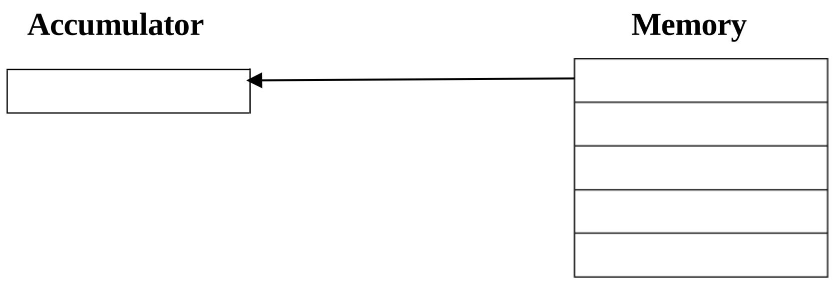
opcode	**operand**

Ex1: LOAD B

This instruction copies the contents of memory location whose symbolic name is given by 'B' into the Accumulator as shown in the figure.

The mathematical representation of this instruction is as follows

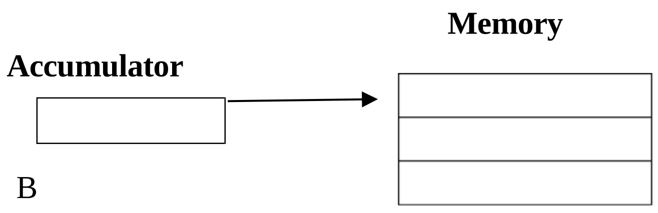
ACC ← [B]



Ex2: STORE B

This instruction copies the contents of Accumulator into memory location whose symbolic name is given by 'B' as shown in the figure. The mathematical representation is as follows

B ← [ACC].



Ex3: ADD B

- This instruction adds the contents of Accumulator with the contents of Memory location 'B' and result is stored in Accumulator.
- The mathematical representation of this instruction is as follows

ACC ←[ACC]+ [B]

STRIGHT LINE SEQUENCING AND INSTRUCTION EXECUTION

Consider the arithmetic expression

C = A+B , Where A,B,C are the memory operands.

The mathematical representation of this instruction is

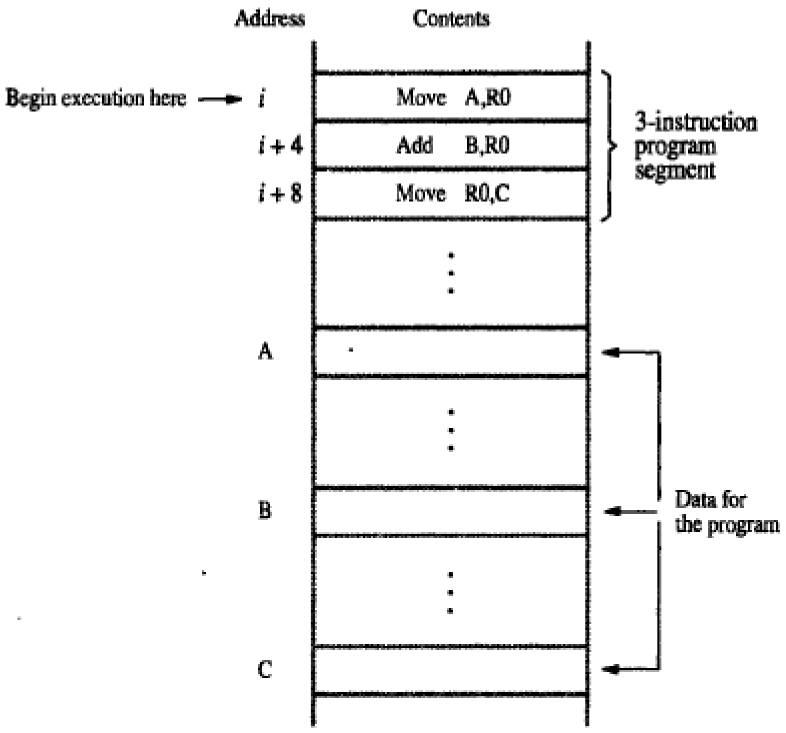
C = [A] + [B].

The sequence of instructions using two address instruction format are as follows

MOV A, R_0
ADD B, R_0
MOV $R_{0,}$ C

Such a program is called as 3 instruction program.
NOTE: The size of each instruction is 32 bits.

- The 3 instruction program is stored in the successive memory locations of the processor is as shown in the fig.



- The system bus consists of uni-directional address bus,bi-directional data bus and control bus

"It is the process of accessing the 1st instruction from memory whose address is stored in program counter into Instruction Register (IR) by means of bi-directional data bus and at the same time after instruction access the contents of PC are incremented by 4 in order to access the next instruction. Such a process is called as "Straight Line Sequencing".

INSTRUCTION EXECUTION

There are 4 steps for instruction execution

1 Fetch the instruction from memory into the Instruction Register (IR) whose address is stored in PC.

$IR \leftarrow [[PC]]$

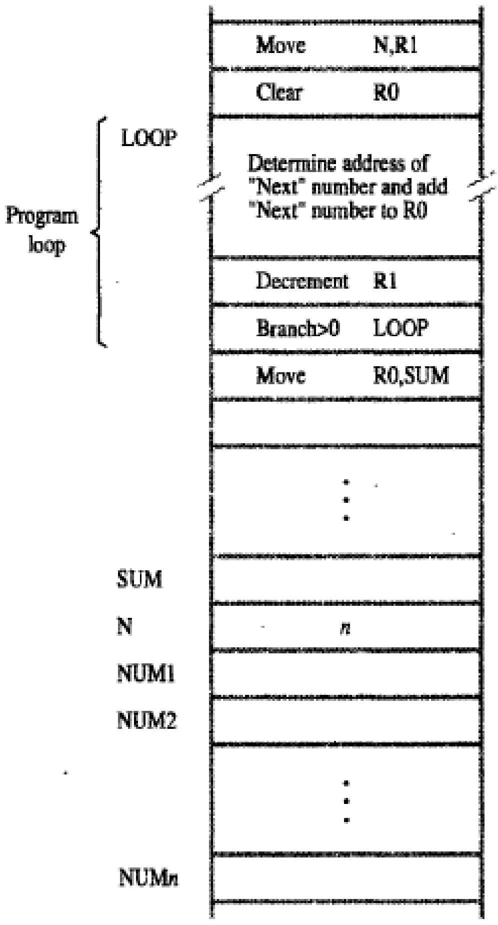
2 Decode the instruction.
3 Perform the operation according to the opcode of an instruction
4 Load the result into the destination.
5 During this process, Increment the contents of PC to point to next instruction (In 32 bit machine increment by 4 address)

PC ← [PC] + 4.

6 The next instruction is fetched, from the address pointed by PC.

BRANCHING

Suppose a list of 'N' numbers have to be added. Instead of adding one after the other, the add statement can be put in a loop. The loop is a straight-line of instructions executed as many times as needed.



The 'N' value is copied to R1 and R1 is decremented by 1 each time in loop. In the loop find the value of next elemet and add it with Ro.

In conditional branch instruction, the loop continues by coming out of sequence only if the condition is true. Here the PC value is set to 'LLOP' if the condition is true.

Branch > 0 LOOP // if >0 go to LOOP

The PC value is set to LOOP, if the previous statement value is >0 ie. after decrementing R1 value is greater than 0.

If R1 value is not greater than 0, the PC value is incremented in a mormal sequential way and the next instruction is executed.

CONDITION CODE BITS

- The processor consists of series of flip-flops to store the status information after ALU operation.
- It keeps track of the results of various operations, for subsequent usage.
- The series of flip-flip-flops used to store the status and control information of the processor is called as “Condition Code Register”. It defines 4 flags. The format of condition code register is as follows.

C	**V**	**Z**	**N**

1 N (NEGATIVE) Flag:
It is designed to differentiate between positive and negative result.
It is set 1 if the result is negative, and set to 0 if result is positive.

2 Z (ZERO) Flag:
It is set to 1 when the result of an ALU operation is found to zero, otherwise it is cleared.

3 V (OVER FLOW) Flag:
In case of 2^s Complement number system n-bit number is capable of representing a range of numbers and is given by -2^{n-1} to $+2^{n-1.}$. The Over-Flow flag is set to 1 if the result is found to be out of this range.

4 C (CARRY) Flag :
This flag is set to 1 if there is a carry from addition or borrow from subtraction, otherwise it is cleared.

8. Addressing Modes

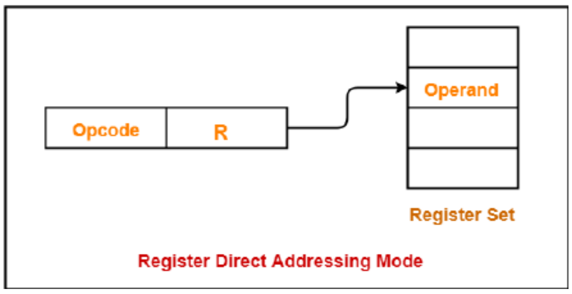
The various formats of representing operand in an instruction or location of an operand is called as “Addressing Mode”. The different types of Addressing Modes are

a) Register Addressing
b) Direct Addressing
c) Immediate Addressing
d) Indirect Addressing
e) Index Addressing
f) Relative Addressing
g) Auto Increment Addressing
h) Auto Decrement Addressing

a. REGISTER ADDRESSING:

In this mode operands are stored in the registers of CPU. The name of the register is directly specified in the instruction.

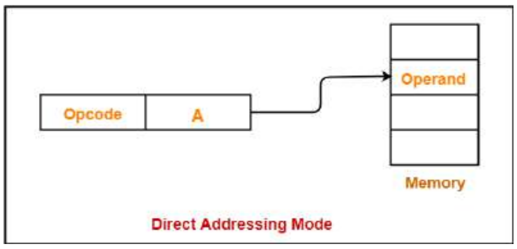
Ex: MOVE R_1,R_2 Where R1 and R2 are the Source and Destination registers respectively. This instruction transfers 32 bits of data from R1 register into R2 register. This instruction does not refer memory for operands. The operands are directly available in the registers.



b. DIRECT ADDRESSING

It is also called as Absolute Addressing Mode. In this addressing mode operands are stored in the memory locations. The name of the memory location is directly specified in the instruction.

Ex: MOVE LOCA, R_1 : Where LOCA is the memory location and R1 is the Register.

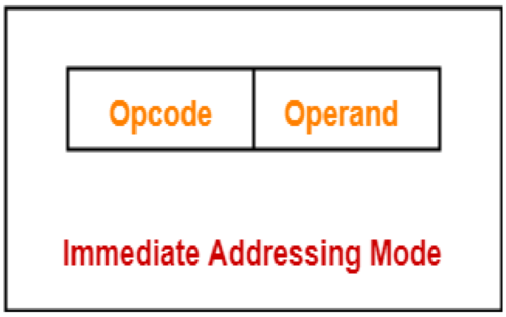


This instruction transfers 32 bits of data from memory location LOCA into the General Purpose Register R1.

c. IMMEDIATE ADDRESSING

In this Addressing Mode operands are directly specified in the instruction. The source field is used to represent the operands. The operands are represented by # (hash) sign.

Ex: MOVE #23, R0



d. INDIRECT ADDRESSING

In this Addressing Mode effective address of an operand is stored in the memory location or General Purpose Register.

[Effective address (EA) – the actual memory address of the operand]

The memory locations or GPRs are used as the memory pointers.

Memory pointer: It stores the address of the memory location.

There are two types Indirect Addressing

i) Indirect through GPRs
ii) Indirect through memory location

i) Indirect Addressing Mode through GPRs

In this Addressing Mode the effective address of an operand is stored in the one of the General Purpose Register of the CPU.

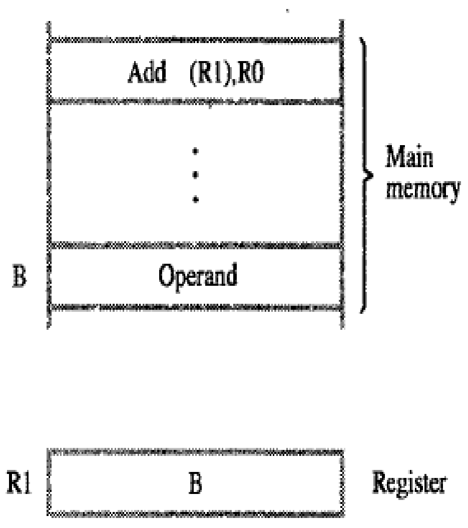
Ex: ADD (R1), R0 ; Where R_1 and R_0 are GPRs

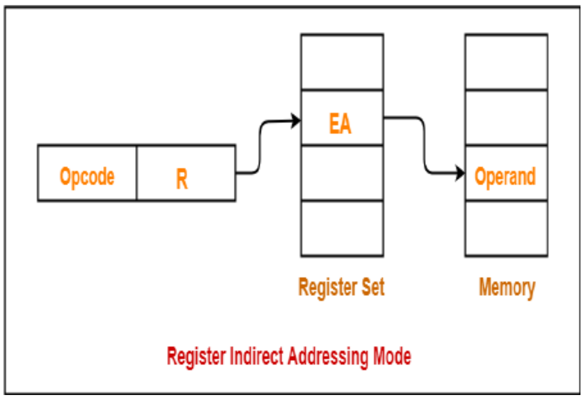
(R1) – R1 stores the address of a location where operand value is present.

This instruction adds the data from the memory location whose address is stored in R_1, with the contents of R_0 Register and the result is stored in R_0 register as shown in the fig.

$R_0 \longleftarrow [[R1]] + R_0$

The diagrammatic representation of this addressing mode is as shown in the fig.





ii) Indirect Addressing Mode through Memory Location.

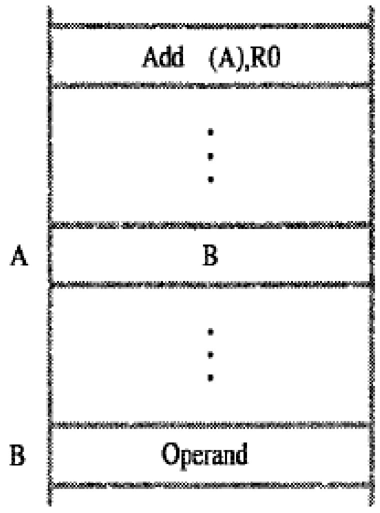
In this Addressing Mode, effective address of an operand is stored in the memory location.

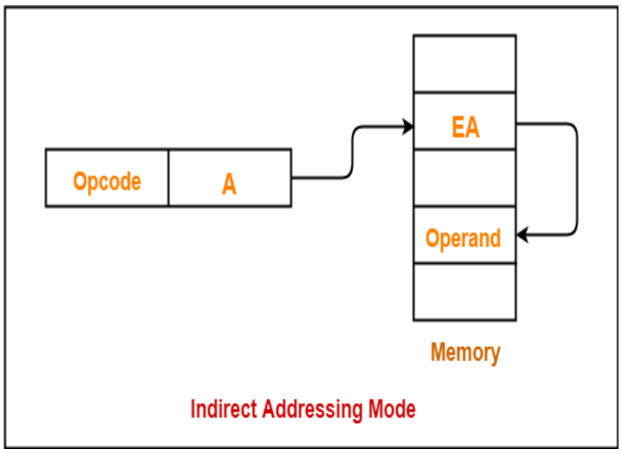
Ex: ADD (A), R_0

This instruction adds the data from the memory location, whose address is stored in 'A' memory location with the contents of R_0 and result is stored in R_0 register.

$R_0 \longleftarrow [[A]] + R_0$

The diagrammatic representation of this addressing mode is as shown in the fig.





e. INDEX ADDRESSING MODE

In this addressing mode, the effective address of an operand is computed by adding constant value with the contents of Index Register. Any one of the General Purpose Register namely R_0 to R_{n-1} can be used as the Index Register. The constant value is directly specified in the instruction.

The symbolic representations of this mode are as follows

1. X (R_i) where X is the Constant value and R_j is the GPR.

   It can be represented as

   Effective Address (EA) of an operand = X + (R_i)

   Eg: Add 5(R2) , R3

   Effective Address(EA) of first operand = 5 + [R2].

2. (R_i , R_J) Where R_i and R_j are the General Purpose Registers used to store addresses of an operand and constant value respectively. It can be represented as

   The EA of an operand is given by EA = (R_i) + (R_j)

3. X (R_i , R_j) Where X is the constant value and R_I and R_J are the General Purpose Registers used to store the addresses of the operands.It can be represented as

The EA of an operand is given by

EA = (R_i) + (R_j) + X

Eg : Add 5(R1)(R2) , R3

EA of first operand is [R1]+[R2]+5

There are two types of Index Addressing Modes

i) Offset is given as constant.

ii) Offset is in Index Register.

Note : Offset : It is the difference between the starting effective address of the memory location and the effective address of the operand fetched from memory.

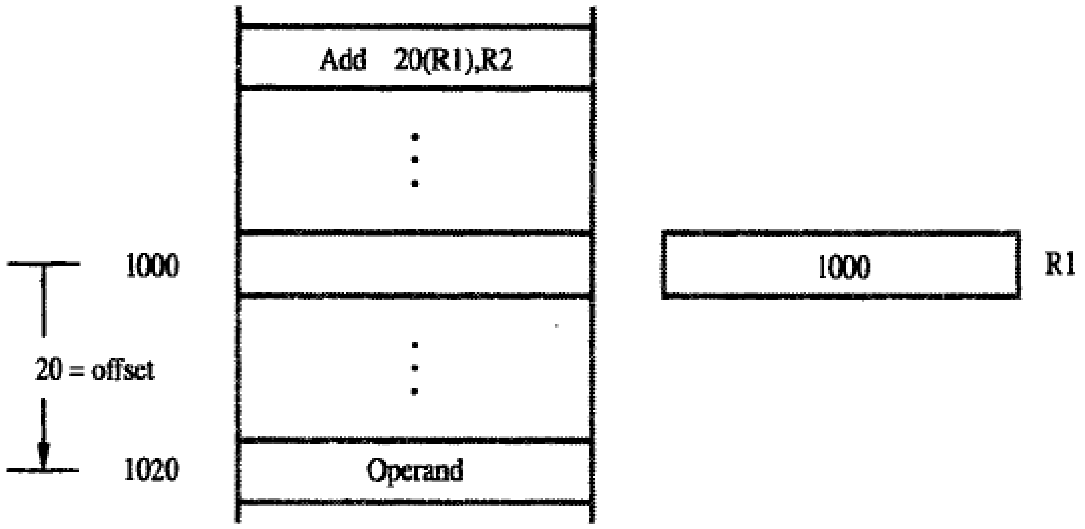
i) Offset is given as constant

Ex: ADD 20(R_1), R_2

The EA of an operand is given by

EA = 20 + [R_1]

This instruction adds the contents of memory location whose EA is the sum of contents of R_1 with 20 and with the contents of R_2 and result is placed in R_2 register. The diagrammatic representation of this mode is as shown in the fig.



ii) Offset is in Index Register

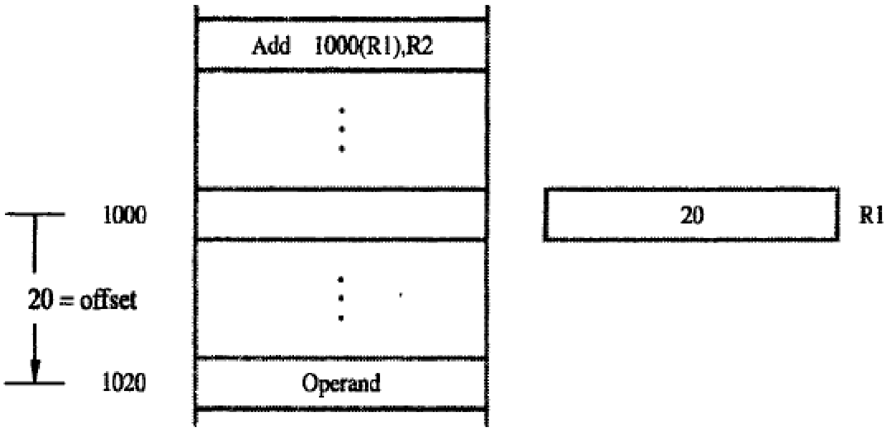
Ex: ADD 1000(R_1) , R_2 R_1 holds the offset address of an operand.

The EA of an operand is given by

EA = 1000 + [R_1]

This instruction adds the data from the memory location whose address is given by [1000 + [R1] with the contents of R_2 and result is placed in R_2 register.

The diagrammatic representation of this mode is as shown in the fig.



f. RELATIVE ADDRESSING MODE:

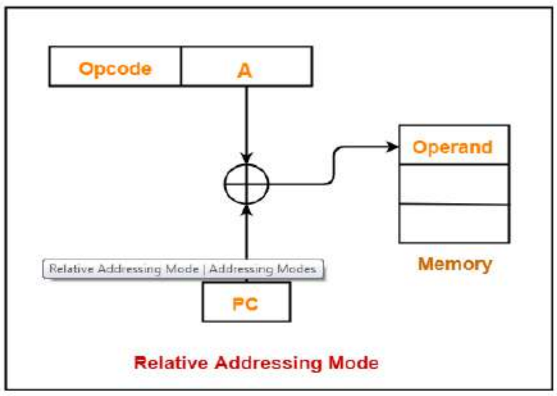
In this Addressing Mode EA of an operand is computed by the Index Addressing Mode. This Addressing Mode uses PC (Program Counter) to store the EA of the next instruction instead of GPR.

The symbolic representation of this mode is X(PC), where X is the offset value and PC is the Program Counter to store the address of the next instruction to be executed.

EA of operand = X + (PC).

This Addressing Mode is useful to calculate the EA of the target memory location.

Effective Address
= Content of Program Counter + Address part of the instruction



g. AUTO INCREMENT ADDRESSING MODE

In this Addressing Mode , EA of an operand is stored in the one of the GPRs of the CPU. This Addressing Mode increment the contents of register, to point to next memory locations after operand access.

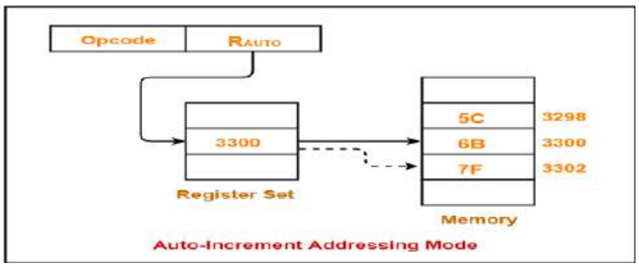
In 32- bit machine, it points to the next memory location, by adding 4 to current location value.

The symbolic representation is

$(R_I)+$ Where R_i is the one of the GPR.

Ex: MOVE $(R_1)+$, R_2

This instruction transfer's data from the memory location whose address is stored in R_1 into R_2 register and then it increments the contents of R_1 to point to next address.



h. AUTO DECREMENT ADDRESSING MODE

In this Addressing Mode , EA of an operand is stored in the one of the GPRs of the CPU. This Addressing Mode decrements the contents of register, to point to previous memory locations after operand access.

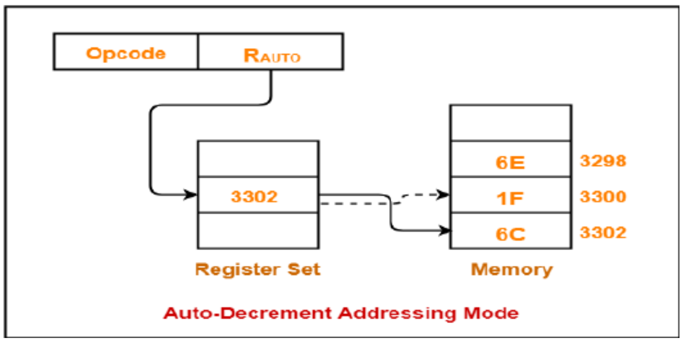
In 32- bit machine, it points to the previous memory location, by subtracting 4 from current location value.

The symbolic representation is

$-(R_I)$ Where R_i is the one of the GPR.

Ex: MOVE $-(R_1)$, R_2

This instruction first decrements the contents of R_1 by 4 memory locations and then transfer's data of that location to destination register.



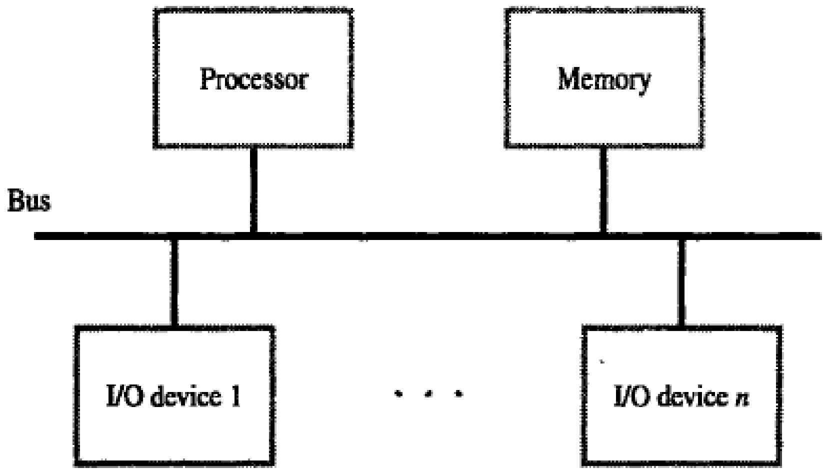
MODULE 4
INPUT/OUTPUT ORGANIZATION

There are a number of input/output (**I/O**) devices, which can be connected to a computer. The input may be from a keyboard, a sensor, switch, mouse etc. Similarly, output may be a speaker, monitor, printer, a digital display etc.

These variety of I/O devices exchange information in varied format, having different word length, transfer speed is different, but are connected to the same system and exchange information with the same computer. Computer must be capable of handling these wide variety of devices.

ACCESSING I/O-DEVICES

A **single bus-structure** can be used for connecting I/O-devices to a computer. The simple arrangement of connecting set of I/O devices to memory and processor by means of system bus is as shown in the figure. Such an arrangement is called as Single Bus Organization.



- The single bus organization consists of
    - Memory
    - Processor
    - System bus
    - I/O device
- The system bus consists of 3 types of buses:

- Address bus (Unidirectional)
- Data bus (Bidirectional)
- Control bus (Bidirectional)

- The system bus enables all the devices connected to it to involve in the data transfer operation.
- The system bus establishes data communication between I/O device and processor.
- Each I/O device is assigned a unique set of address.
- When processor places an address on address-lines, the intended-device responds to the command.
- The processor requests either a read or write-operation.
- The requested data are transferred over the data-lines

Steps for input operation:

- The address bus of system bus holds the address of the input device.
- The control unit of CPU generates $\overline{\text{IORD}}$ Control signal.
- When this control signal is activated the processor reads the data from the input device (DATAIN) into the CPU register.

Steps for output operation:

- The address bus of system bus holds the address of the output device.
- The control unit of CPU generates $\overline{\text{IOWR}}$ control signal.
- When this control signal is enabled CPU transfers the data from processor register to output device(DATAOUT)

There are 2 schemes available to connect I/O devices to CPU

1. **Memory mapped I/O:**

- In this technique, both memory and I/O devices can share the common memory to store the data, the I/O instructions are mapped to any memory location.
- All memory related instructions are used for data transfer between I/O and processor.
- In case of memory mapped I/O input operation can be implemented as,

$$\begin{tikzcd} \text{MOVE} & \text{DATAIN ,} \arrow[d] & \text{R0} \arrow[d] \\ & \text{Source} & \text{destination} \end{tikzcd}$$

This instruction sends the contents of location DATAIN to register R0.

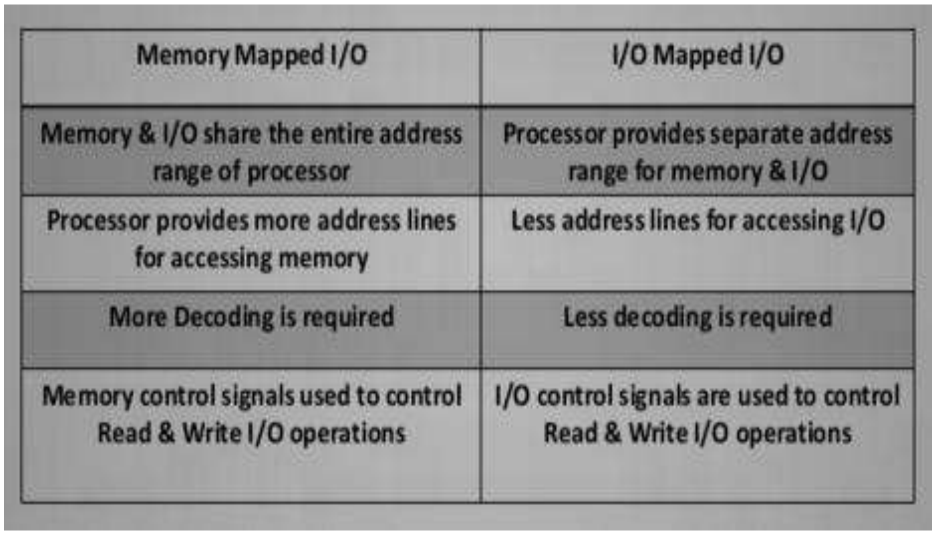
- Similarly output can be implemented as,

$$\begin{tikzcd}[row sep=small] \text{MOVE} & \text{R0,} \arrow[d] & \text{DATAOUT} \arrow[d] \\ & \text{Source} & \text{destination} \end{tikzcd}$$

The data is written from R0 to DATAOUT location (address of output buffer)

2. I/O Mapped I/O:

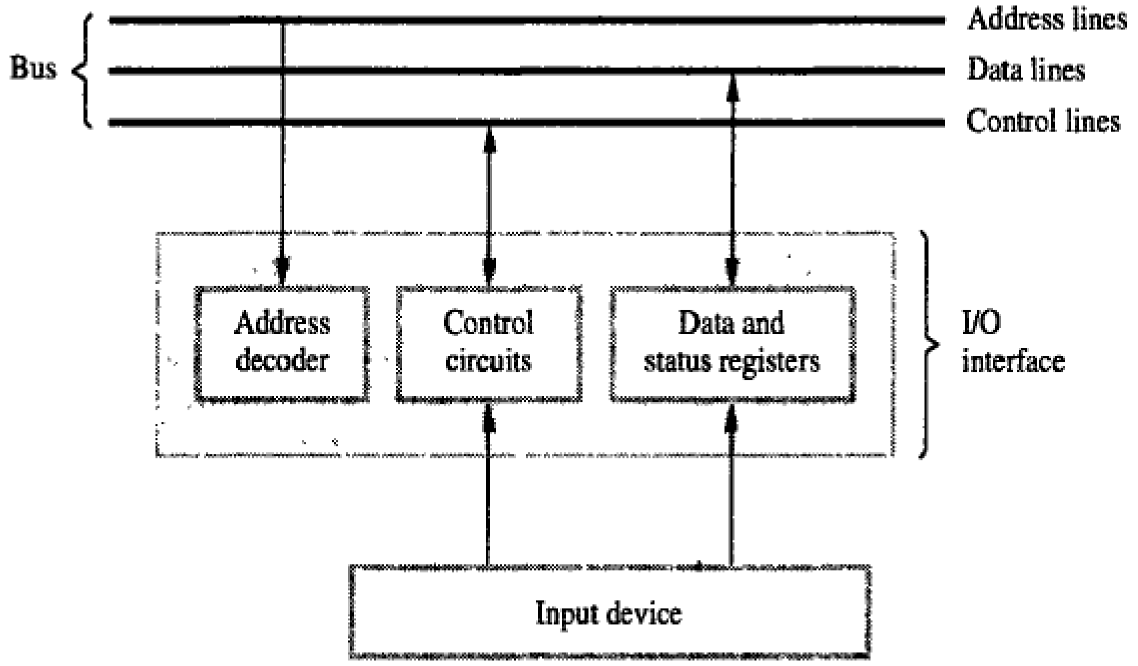
- In this technique, a separate address space is allocated for I/O devices. Address space for program and I/O devices are different.
- Hence two sets of instruction are used for data transfer.
- One set for memory operations and another set for I/O operations.
- Whole address space is available for the program.
- Eg – IN AL, DX



Memory Mapped I/O	I/O Mapped I/O
Memory & I/O share the entire address range of processor	Processor provides separate address range for memory & I/O
Processor provides more address lines for accessing memory	Less address lines for accessing I/O
More Decoding is required	Less decoding is required
Memory control signals used to control Read & Write I/O operations	I/O control signals are used to control Read & Write I/O operations

I/O INTERFACE

The hardware arrangement of connecting i/p device to the system bus is as shown in the fig.



This hardware arrangement is called as I/O interface. The I/O interface consists of 3 functional devices namely:

1) **Address Decoder:**
   - Its function is to decode the address, in-order to recognize the input device whose address is available on the unidirectional address bus.
   - The recognition of input device is done first, and then the control and data registers becomes active.
   - The unidirectional address bus of system bus is connected to input of the address decoder as shown in figure

2) **Control Circuit:**
   - The control bus of system bus is connected to control circuit as shown in the fig.
   - The processor sends commands to the I/O system through the control bus.
   - It controls the read write operations with respect to I/O device.

3) **Status & Data register:**
   - It specifies type of operation (either read or write operation) to be performed on I/O device. It specifies the position of operation.

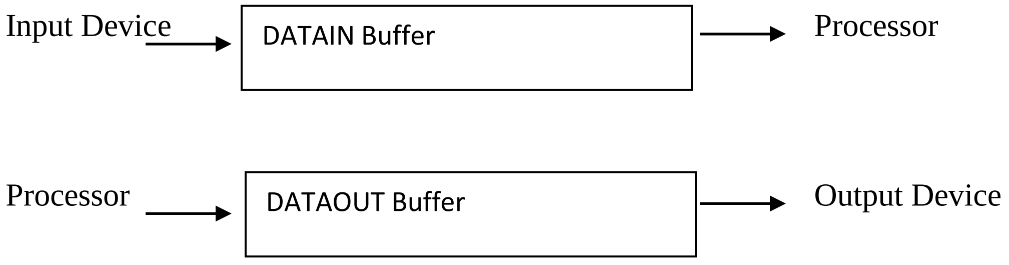
4) **Data Register:**

- The data bus carries the data from the I/O devices to or from the processor. The data bus is connected to the data/ status register.
- The data register stores the data, read from input device or the data, to be written into output device. There are 2 types:

  DATAIN - Input-buffer associated with keyboard.

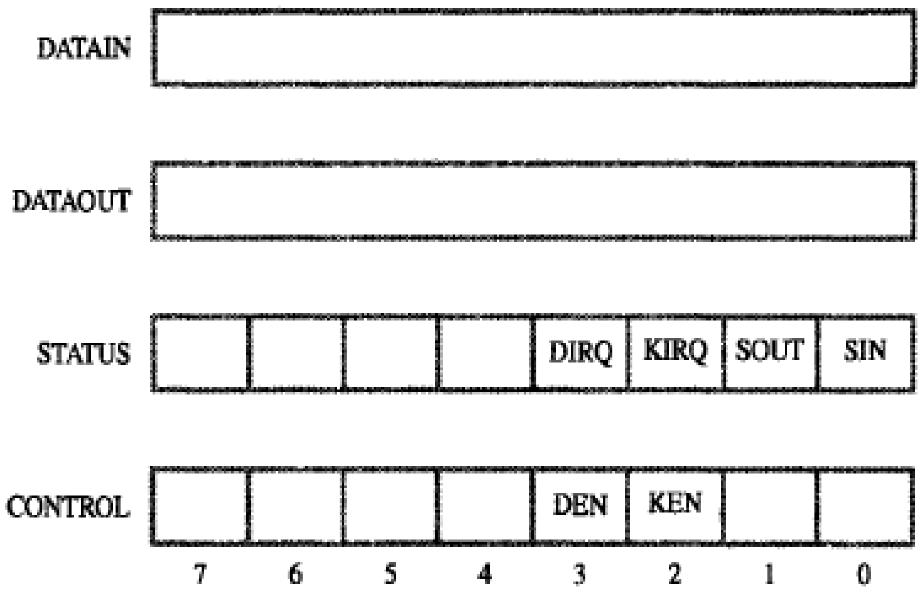
  DATAOUT -Output data buffer of a display/printer.

Data buffering is an essential task of an I/O interface. Data transfer rates of processor and memory are high, when compared with the I/O devices, hence the data are buffered at the I/O interface circuit and then forwarded to output device, or forwarded to processor in case of input devices.



Input & Output registers –

Various registers in keyboard and display devices -



DATAIN register is a part of input device. It is used to store the ASCII characters read from keyboard.

DATAOUT register is a part of output device. It is used to store the ASCII characters to be displayed on the output device.

STATUS register stores the status of working of I/O devices –

- SIN flag – This flag is set to 1, when DATAIN buffer contains the data from keyboard. The flag is set to 0, after the data is passed from DATAIN buffer to the processor.
- SOUT flag – This flag is set to 1, when DATAOUT buffer is empty and the data can be added to it by processor. The flag is set to 0, when DATAOUT buffer has the data to be displayed.
- KIRQ (Keyboard Interrupt Request) – By setting this flag to 1, keyboard requests the processor to obtain its service and an interrupt is sent to the processor. It is used along with the SIN flag.
- DIRQ(Display Interrupt Request) – The output device request the processor to obtain its service for output operation, by activating this flag to 1.

Control registers
KEN (keyboard Enable) – Enables the keyboard for input operations.
DEN (Display Enable) – Enables the output device for input operations.

Program Controlled I/O

- It is the process of controlling the input and output operations by executing 2 sets of instruction, one set for input operation and the next set for output operation.
- The program checks the status of I/O register and reads or displays data. Here the I/O operation is controlled by program.

```
WAITK       TestBit #0, STATUS       (Checks SIN flag)
            Branch = 0 WAITK
            Move DATAIN, R0            (Read character)
```
[*Code to read a character from DATAIN to R0]

This code checks the SIN flag, and if it is set to 0 (ie. If no character in DATAIN Buffer), then move back to WAITK label. This loop continues until SIN flag is set to 1. When SIN is 1, data is moved from DATAIN to R0 register. Thus the program, continuously checks for input operation.

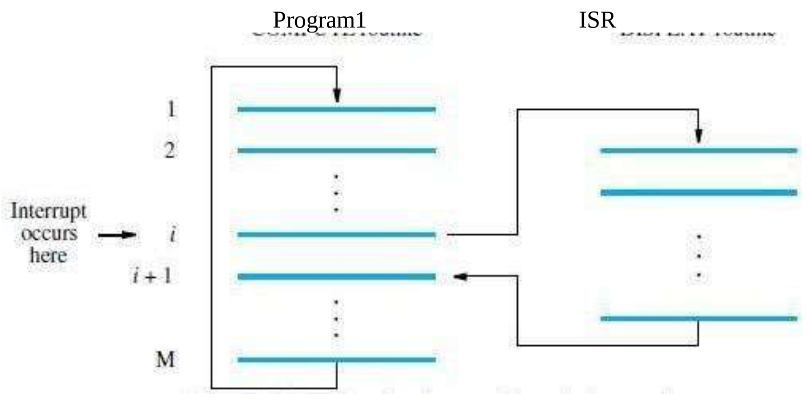
Similarly code for Output operation,

```
WAITD       TestBit #0, STATUS         (Checks SOUT flag)
            Branch = 0 WAITD
            Move R0, DATAOUT           (Send character for display)
```

The code checks the SOUT flag, and if it is set to 1 (ie. If no character in DATAOUT Buffer), then move back to WAITK label. This loop continues until SOUT flag is set to 0. When SOUT is 0, data is moved from R0 register to DATAOUT (ie. Sent by processor).

Interrupt

- It is an event which suspends the execution of one program and begins the execution of another program.
- In program controlled I/O, a program should continuously check whether the I/O device is free. By this continuous checking the processor execution time is wasted. It can be avoided by I/O device **sending an 'interrupt' to the processor, when I/O device is free**.
- The interrupt invokes a subroutine called **Interrupt Service Routine (ISR**), which resolves the cause of interrupt.
- The occurrence of interrupt causes the processor to transfer the execution control from user program to ISR.



The following steps takes place when the interrupt related instruction is executed:

- After the execution of current instruction i.
- Transfer the execution control to sub program from main program.
- Increments the content of PC by 4 memory location.
- It decrements SP by 4 memory locations.
- Pushes the contents of PC into the stack segment memory whose address is stored in SP.
- It loads PC with the address of the first instruction of the sub program.

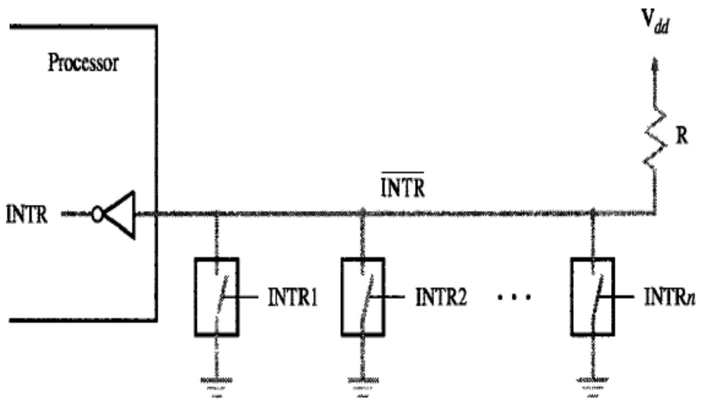
The following steps takes place when '*return*' instruction is executed in ISR -

- It transfers the execution control from ISR to user program.
- It retrieves the content of stack memory location whose address is stored in SP into the PC.
- After retrieving the return address from stack memory location into the PC it increments the Content of SP by 4 memory location.

Interrupt Latency / interrupt response time is the delay between the time taken for receiving an interrupt request and start of the execution of the ISR.Generally, the long interrupt latency in unacceptable.

INTERRUPT HARDWARE

- The external device (I/O device) sends interrupt request to the processor by activating a bus line and called as interrupt request line.
- All I/O device uses the same single interrupt-request line.
- One end of this interrupt request line is connected to input power supply by means of a register.
- The another end of interrupt request line is connected to INTR (Interrupt request) signal of processor as shown in the fig.



- The I/O device is connected to interrupt request line by means of switch, which is grounded as shown in the fig.
- When all the switches are open the voltage drop on interrupt request line is equal to the V_{DD} and INTR value at process is 0.
- This state is called as in-active state of the interrupt request line.

- The I/O device interrupts the processor by closing its switch.
- When switch is closed the voltage drop on the interrupt request line is found to be zero, as the switch is grounded, hence $\overline{\text{INTR}}$=0 and INTR=1.
- The signal on the interrupt request line is logical OR of requests from the several I/O devices. Therefore, $\overline{\text{INTR}}=\overline{\text{INTR1}} + \overline{\text{INTR2}} + \ldots\ldots\ldots\ldots + \overline{\text{INTRn}}$

ENABLING AND DISABLING THE INTERRUPTS

The arrival of interrupt request from external devices or from within a process, causes the suspension of on-going execution and start the execution of another program.

- Interrupt arrives at any time and it alters the sequence of execution. Hence the interrupt to be executed must be selected carefully.
- All computers can enable and disable interruptions as desired.
- When an interrupt is under execution, other interrupts should not be invoked. This is performed in a system in different ways.
- The problem of infinite loop occurs due to successive interruptions of active INTR signals.
- There are 3 mechanisms to solve problem of infinite loop:
  1) Processor should ignore the interrupts until execution of first instruction of the ISR.
  2) Processor should automatically disable interrupts before starting the execution of the ISR.
  3) Processor has a special INTR line for which the interrupt-handling circuit.

     Interrupt-circuit responds only to leading edge of signal. Such line is called edge-triggered.
- Sequence of events involved in handling an interrupt-request:
  1) The device raises an interrupt-request.
  2) The processor interrupts the program currently being executed.
  3) Interrupts are disabled by changing the control bits in the processor status register (PS).
  4) The device is informed that its request has been recognized.

     In response, the device deactivates the interrupt-request signal.
  5) The action requested by the interrupt is performed by the interrupt-service routine.
  6) Interrupts are enabled and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

While handling multiple devices, the issues concerned are:

- How can the processor recognize the device requesting an interrupt?
- How can the processor obtain the starting address of the appropriate ISR?
- Should a device be allowed to interrupt the processor while another interrupt is being serviced?
- How should 2 or more simultaneous interrupt-requests be handled?

VECTORED INTERRUPT

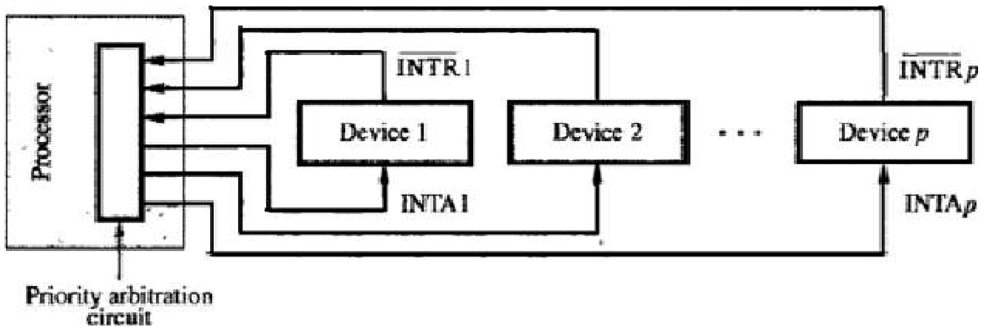
- A device requesting an interrupt identifies itself by sending a special-code to processor over bus.
- Then, the processor starts executing the ISR.
- The special-code indicates starting-address of ISR.
- The special-code length ranges from 4 to 8 bits.
- The location pointed to by the interrupting-device is used to store the staring address to ISR.
- The staring address to ISR is called the **interrupt vector**.
- Processor

  → loads interrupt-vector into PC &

  → executes appropriate ISR.
- When processor is ready to receive interrupt-vector code, it activates INTA line.
- Then, I/O-device responds by sending its interrupt-vector code & turning off the INTR signal.
- The interrupt vector also includes a new value for the Processor Status Register

INTERRUPT NESTING

- A multiple-priority scheme is implemented by using separate INTR & INTA lines for each device
- Each INTR line is assigned a different priority-level as shown in Figure.


Processor
$\overline{\text{INTR}}1$
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA1
INTAp
Priority arbitration circuit

- Priority-level of processor is the priority of program that is currently being executed.
- Processor accepts interrupts only from devices that have higher-priority than its own.
- At the time of execution of ISR for some device, priority of processor is raised to that of the device.
- Thus, interrupts from devices at the same level of priority or lower are disabled.

Privileged Instruction

- Processor's priority is encoded in a few bits of PS word. (PS = Processor-Status).
- Encoded-bits can be changed by **Privileged Instructions** that write into PS.
- Privileged-instructions can be executed only while processor is running in **Supervisor Mode**.
- Processor is in supervisor-mode only when executing operating-system routines.

Privileged Exception

- User program cannot

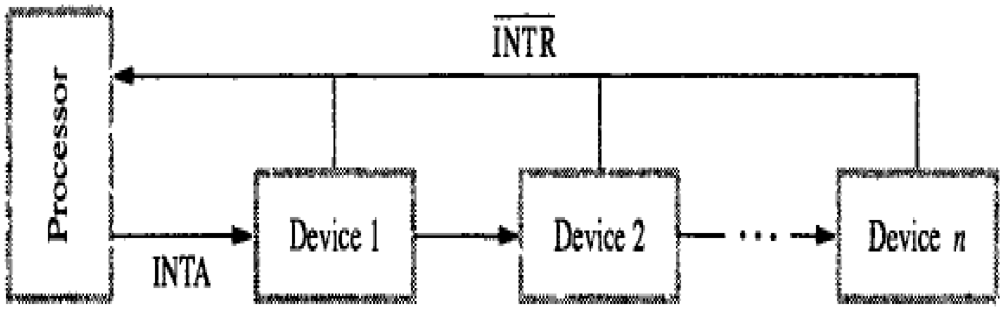
  → accidently or intentionally change the priority of the processor &

  → disrupt the system-operation.
- An attempt to execute a privileged-instruction while in user-mode leads to a **Privileged Exception**.

SIMULTANEOUS REQUESTS

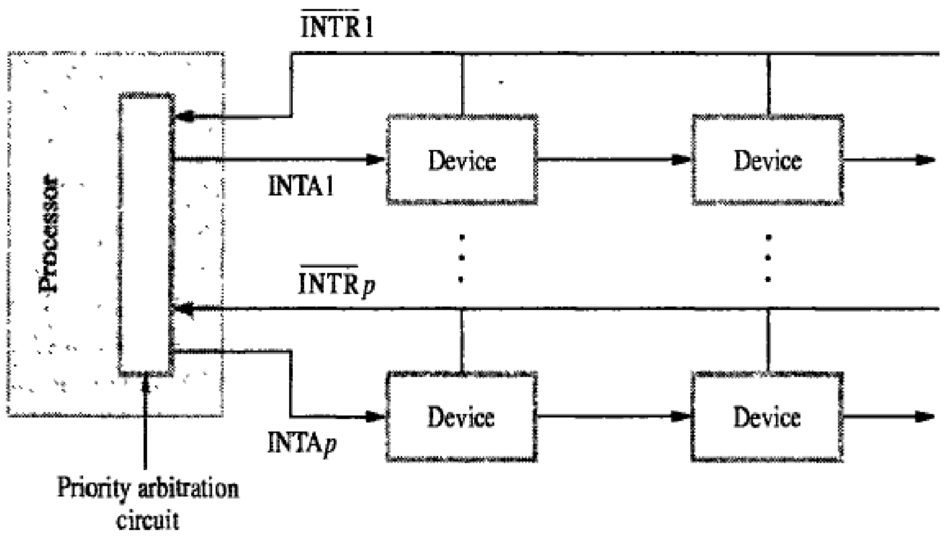
DAISY CHAIN

- The daisy chain with multiple priority levels is as shown in the figure.



- The interrupt request line INTR is common to all devices as shown in the fig.
- The interrupt acknowledge line is connected in a daisy fashion as shown in the figure.
- This signal propagates serially from one device to another device.
- The several devices raise an interrupt by activating INTR signal. In response to the signal, processortransfers its device by activating INTA signal.
- This signal is received by device 1. The device-1 blocks the propagation of INTA signal to device-2,when it needs processor service.
- The device-1 transfers the INTA signal to next device when it does not require the processor service.
- In daisy chain arrangement device-1 has the highest priority.
- **Advantage:** It requires fewer wires than the individual connections.

ARRANGEMENT OF PRIORITY GROUPS


$\overline{\text{INTR}}1$
Processor
Device
Device
INTA1
$\overline{\text{INTR}}p$
Device
Device
INTAp
Priority arbitration circuit

- In this technique, devices are organizes in a group and each group is connected to the processor at a different priority level.
- Within a group device are connected in a daisy chain fashion as shown in the figure.

Direct Memory Access (DMA)

- Direct Memory Access is the process of transferring the block of data at high speed in between main memory and external device (I/O devices) without continuous intervention of CPU.
- This operation is performed by the control circuit, called as DMA controller.
- DMA controller is a part of the I/O interface.
- The data transfer operation in DMA is processed by the help of DMA controller.
- To initiate Directed data transfer between main memory and external devices DMA controller needs parameters from the CPU.
- These 3 Parameters are:

1) Starting address of the memory block.
2) No of words to be transferred.
3) Type of operation (Read or Write).

After receiving these 3 parameters from CPU, DMA controller establishes directed data transfer operation between main memory and external devices without the involvement of CPU. Hence the processor is free to execute other programs.

Register of DMA Controller:

It consists of 3 type of register:

Starting address register:

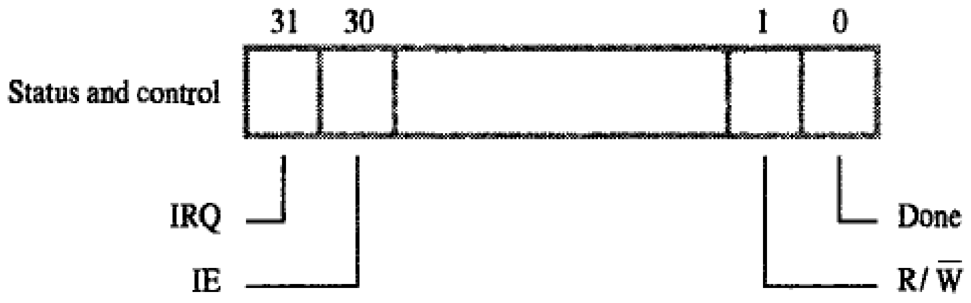
The format of starting address register is as shown in the fig. It is used to store the starting address of the memory block.

Word-Count register:

The format of word count register is as shown in fig. It is used to store the no of words to be transferred from main memory to external devices and vice versa.

Status and Controller register:

The format of status and controller register is as shown in fig.



a) DONE bit:

- The DMA controller sets this bit to 1 when it completes the direct data transfer between main memory and external devices.
- This information is informed to CPU by means of DONE bit.

b) $R/\overline{W}$ (Read or Write):

- This bit is used to differentiate between memory read and memory write operation. It is set by a program instruction.
- The $R/\overline{W}$ = 1 for read operation and
  = 0 for write operation.
- When this bit is set to 1, DMA controller performs read operation and transfers one block of data from main memory to external device.
- When this bit is set to 0, DMA controller performs write operation and transfers one block of data from external device to main memory.

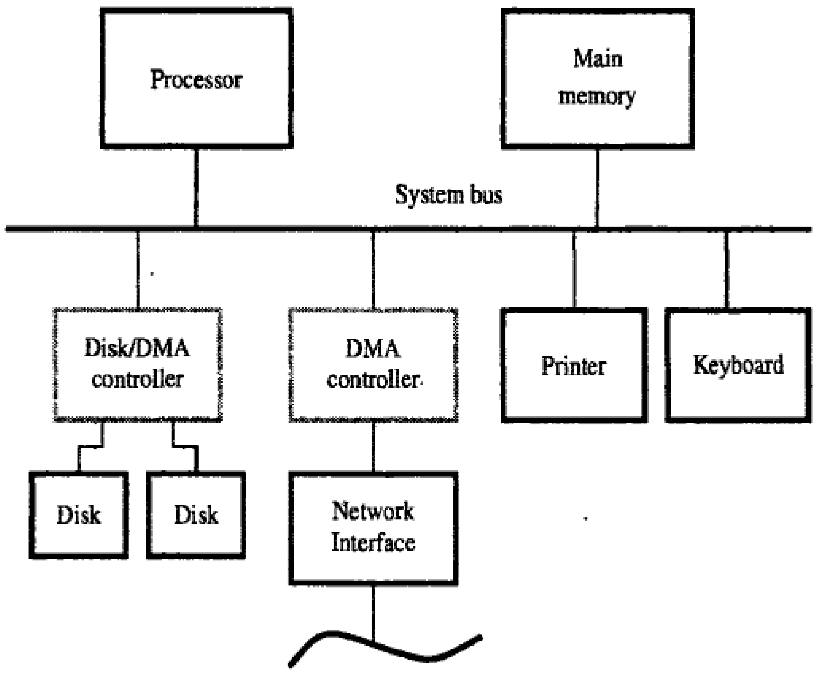
c) IE (Interrupt enable) bit:

- The DMA controller enables the interrupt enable bit after the completion of DMA operation

d)**Interrupt request (IRQ):**

- The DMA controller requests the CPU for permission and data, to transfer new block of data from source to destination by activating this bit.

The computer with DMA controller is as shown in the fig.:



- In the sample architecture shown above, the DMA controller connects two external devices namely disk 1 and disk 2 to system bus.
- The DMA controller also interconnects high speed network devices to system bus as shown in the above fig.
- Let us consider direct data transfer operation by means of DMA controller without the involvement of CPU in between main memory and disk 1.
- To establish direct data transfer operation between main memory and disk 1. DMA controller request the processor to obtain 3 parameters namely:
    1) Starting address of the memory block.
    2) No of words to be transferred.
    3) Type of operation (Read or Write).

- After receiving these 3 parameters from processor, DMA controller directly transfers block of data between main memory and external devices (disk 1) depending on the operation.
- This information is informed to CPU by setting respective bits in the status and controller register of DMA controller.
  These are 2 types of request with respect to system bus
  1). CPU request.
  2). DMA request.
  Highest priority will be given to DMA request.

- Actually the CPU generates memory cycles to perform read and write operations.
  The DMA controller steals memory cycles from the CPU to perform read and write operations. This approach is called as **"Cycle stealing".**

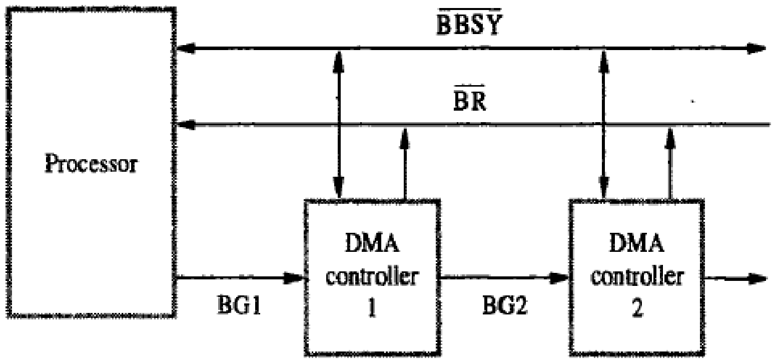
- An exclusive option will be given for DMA controller to transfer block of data between external devices and main memory. Only after the transfer of whole block, signal is sent to the processor. This technique is called as "**Burst mode of operation.**"
- Conflict may arise, if CPU and multiple DMA controllers, request for bus, at the same time. This is resolved by bus arbitration.

BUS ARBITRATION

- Any device which initiates data transfer operation on bus at any instant of time is called as Bus-Master.
- When the bus mastership is transferred from one device to another device, the next device is ready to obtain the bus mastership.
- The bus-mastership is transferred from one device to another device based on the principle of priority system. There are two types of bus-arbitration technique:

a)Centralized bus arbitration:

In this technique CPU acts as a bus-master or any control unit connected to bus can be acts as a bus master.



The schematic diagram of centralized bus arbitration is as shown in the fig.:

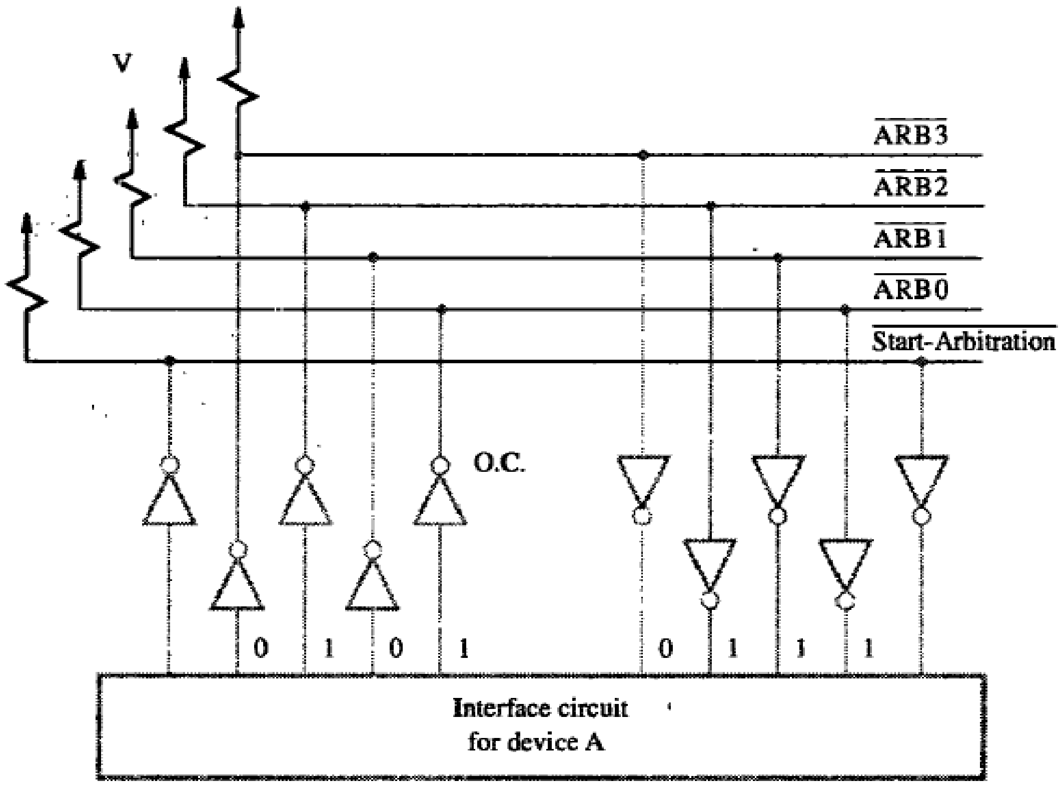
The following steps are necessary to transfer the bus mastership from CPU to one of the DMA controller:

- The DMA controller request the processor to obtain the bus mastership by activating $\overline{BR}$ (Bus request) signal
- In response to this signal the CPU transfers the bus mastership to requested devices DMA controller1 in the form of BG (Bus grant).
- When the bus mastership is obtained from CPU the DMA controller1 blocks the propagation of bus grant signal from one device to another device.
- The $\overline{BG}$ signal is connected to DMA controller2 from DMA controller1, and so on as in daisy fashion style as shown in the figure.
- When the DMA controller1 has not sent BR request, it transfers the bus mastership to DMA controller2 by unblocking bus grant signal.

- When the DMA controller1 receives the bus grant signal, it blocks the signal from passing to DMA controller2 and enables BBSY signal. When BBSY signal is set to 1 the set of devices connected to system bus doesn't have any rights to obtain the bus mastership from the CPU.

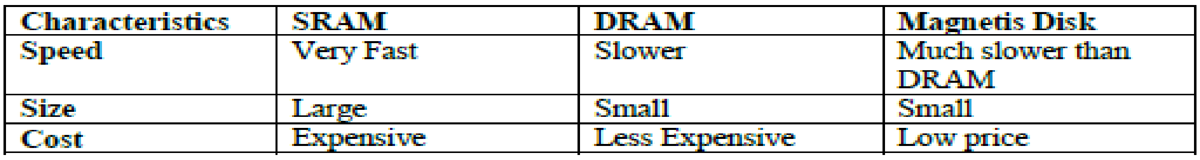
b)Distributed bus arbitration:

- In this technique 2 or more devices trying to access system bus at the same time may participate in bus arbitration process.
- The schematic diagram of distributed bus arbitration is as shown in the figure:

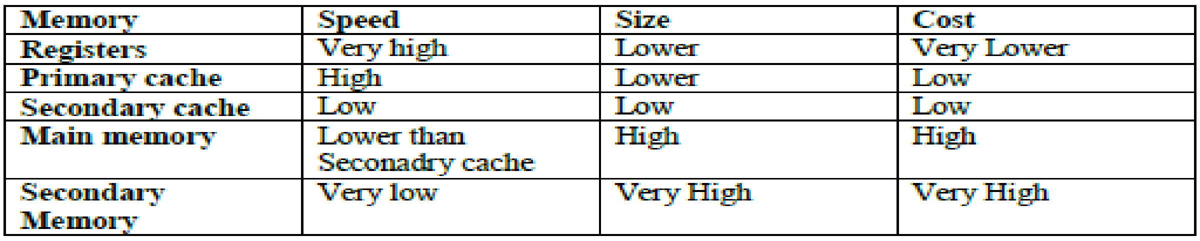


- The external device requests the processor to obtain bus mastership by enabling start arbitration signal.
- In this technique 4 bit code is assigned to each device to request the CPU in order to obtain bus mastership.
- Two or more devices request the bus by placing 4 bit code over the system bus.
- The signals on the bus interpret the 4 bit code and produces winner as a result from the CPU.
- When the input to the one driver = 1, and input to the another driver = 0, on the same bus line, this state is called as "Low level voltage state of bus".
- Consider 2 devices namely A & B trying to access bus mastership at the same time. Let assigned code for devices A & B are 5 (0101) & 6 (0110) respectively.
- The device A sends the pattern (0101) and device B sends its pattern (0110) to master. The signals on the system bus interpret the 4 bit code for devices A & B produces device B as a winner.
- The device B can obtain the bus mastership to initiate direct data transfer between external devices and main memory.

SPEED, SIZE COST



Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price



Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

- The main-memory can be built with DRAM (Figure 8.14)
- Thus, SRAM‟s are used in smaller units where speed is of essence.
- The Cache-memory is of 2 types:

1) Primary/Processor Cache (Level1 or L1 cache)

➢ It is always located on the processor-chip.

2) Secondary Cache (Level2 or L2 cache)

➢ It is placed between the primary-cache and the rest of the memory.

- The memory is implemented using the dynamic components (SIMM, RIMM, DIMM).
- The access time for main-memory is about 10 times longer than the access time for L1 cache.

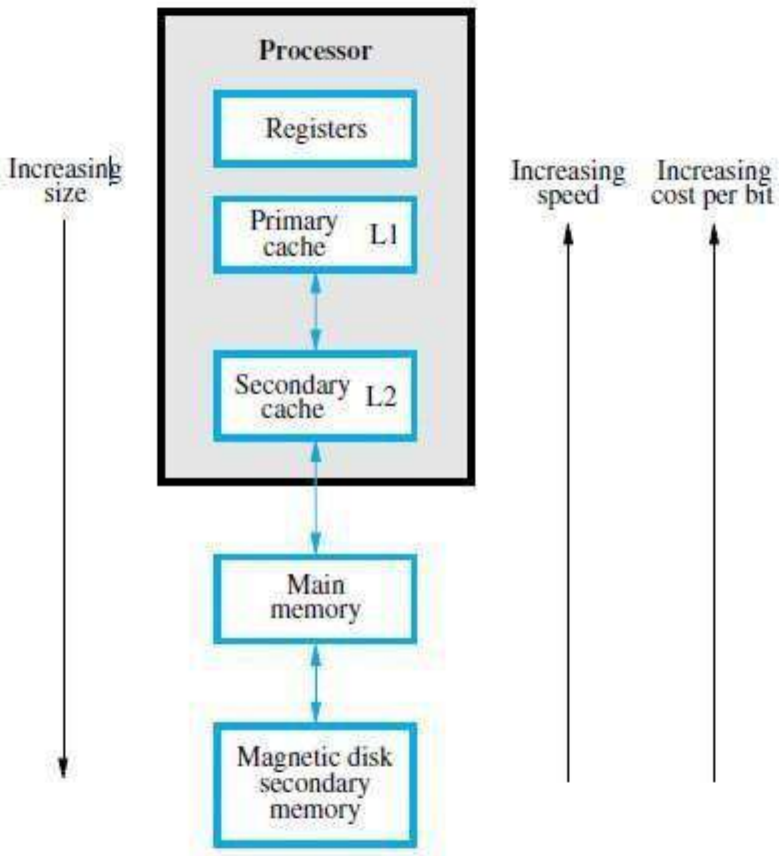


Figure 8.14 Memory hierarchy.

CACHE MEMORIES

• The effectiveness of cache mechanism is based on the property of '**Locality of Reference'.**

Locality of Reference

• Many instructions in the localized areas of program are executed repeatedly during some time period

• Remainder of the program is accessed relatively infrequently (Figure 8.15).

• There are 2 types:

1) <u>Temporal</u>

➢ The recently executed instructions are likely to be executed again very soon.

2) <u>Spatial</u>

➢Instructions in close proximity to recently executed instruction are also likely to be executed soon.

• If active segment of program is placed in cache-memory, then total execution time can be reduced.

• **Block** refers to the set of contiguous address locations of some size.

• The cache-line is used to refer to the cache-block.

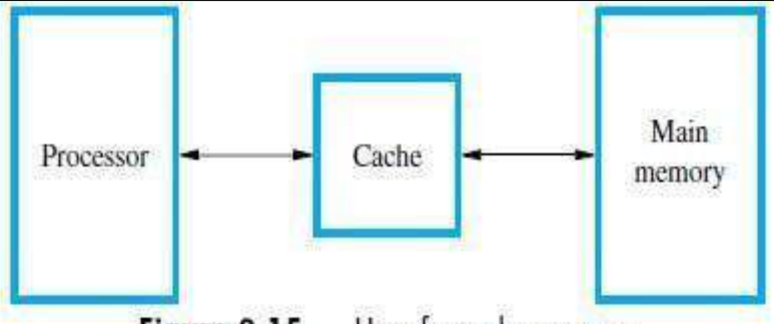


Figure 8.15 Use of a cache memory.

- The Cache-memory stores a reasonable number of blocks at a given time.
- This number of blocks is small compared to the total number of blocks available in main-memory.
- Correspondence b/w main-memory-block & cache-memory-block is specified by mapping-function.
- Cache control hardware decides which block should be removed to create space for the new block.
- The collection of rule for making this decision is called the **Replacement Algorithm.**
- The cache control-circuit determines whether the requested-word currently exists in the cache.
- The write-operation is done in 2 ways: 1) Write-through protocol & 2) Write-back protocol.

Write-Through Protocol

➢ Here the cache-location and the main-memory-locations are updated simultaneously.

Write-Back Protocol

➢ This technique is to
→ update only the cache-location &
→ mark the cache-location with associated flag bit called **Dirty/Modified Bit.**

➢ The word in memory will be updated later, when the marked-block is removed from cache.

During Read-operation

- If the requested-word currently not exists in the cache, then **read-miss** will occur.
- To overcome the read miss, *Load–through/Early restart protocol* is used.

Load–Through Protocol

➢ The block of words that contains the requested-word is copied from the memory into cache.

➢ After entire block is loaded into cache, the requested-word is forwarded to processor.

During Write-operation

- If the requested-word not exists in the cache, then **write-miss** will occur.

1) If **Write Through Protocol** is used, the information is written directly into main-memory.

2) If **Write Back Protocol** is used,
→ then block containing the addressed word is first brought into the cache &

→ then the desired word in the cache is over-written with the new information.

MAPPING FUNCTIONS

- Here we discuss about 3 different mapping-function:
  1) Direct Mapping
  2) Associative Mapping
  3) Set-Associative Mapping

DIRECT MAPPING

- The block-j of the main-memory maps onto block-j modulo-128 of the cache (Figure 8.16).
- When the memory-blocks 0, 128, & 256 are loaded into cache, the block is stored in cache-block 0.

  Similarly, memory-blocks 1, 129, 257 are stored in cache-block 1.
- The contention may arise when
  1) When the cache is full.
  2) When more than one memory-block is mapped onto a given cache-block position.
- The contention is resolved by allowing the new blocks to overwrite the currently resident-block.
- Memory-address determines placement of block in the cache.

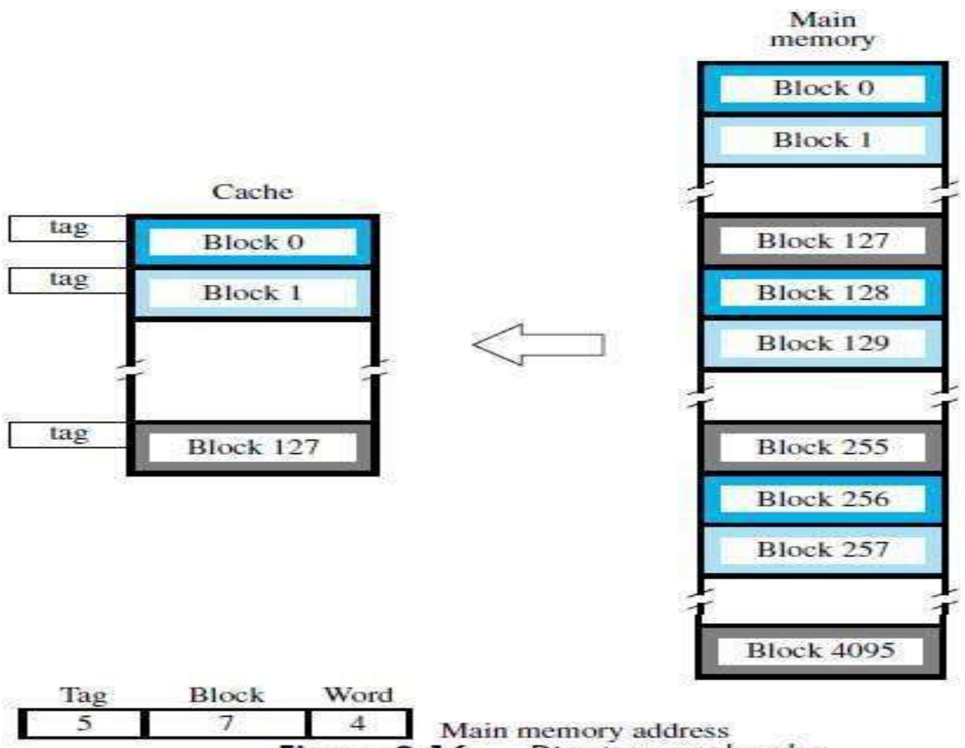


Figure 8.16 Direct-mapped cache.

- The memory-address is divided into 3 fields:

1) **<u>Low Order 4 bit field</u>**

➢ Selects one of 16 words in a block.

2) **<u>7 bit cache-block field</u>**

➢ 7-bits determine the cache-position in which new block must be stored.

3) **<u>5 bit Tag field</u>**

➢ 5-bits memory-address of block is stored in 5 tag-bits associated with cache-location.

➢ As execution proceeds, 5-bit tag field of memory-address is compared with tag-bits associated with cache-location. If they match, then the desired word is in that block of the cache. Otherwise, the block containing required word must be first read from the memory. And then the word must be loaded into the cache.

ASSOCIATIVE MAPPING

- The memory-block can be placed into any cache-block position. (Figure 8.17).
- 12 tag-bits will identify a memory-block when it is resolved in the cache.
- Tag-bits of an address received from processor are compared to the tag-bits of each block of cache.
- This comparison is done to see if the desired block is present.

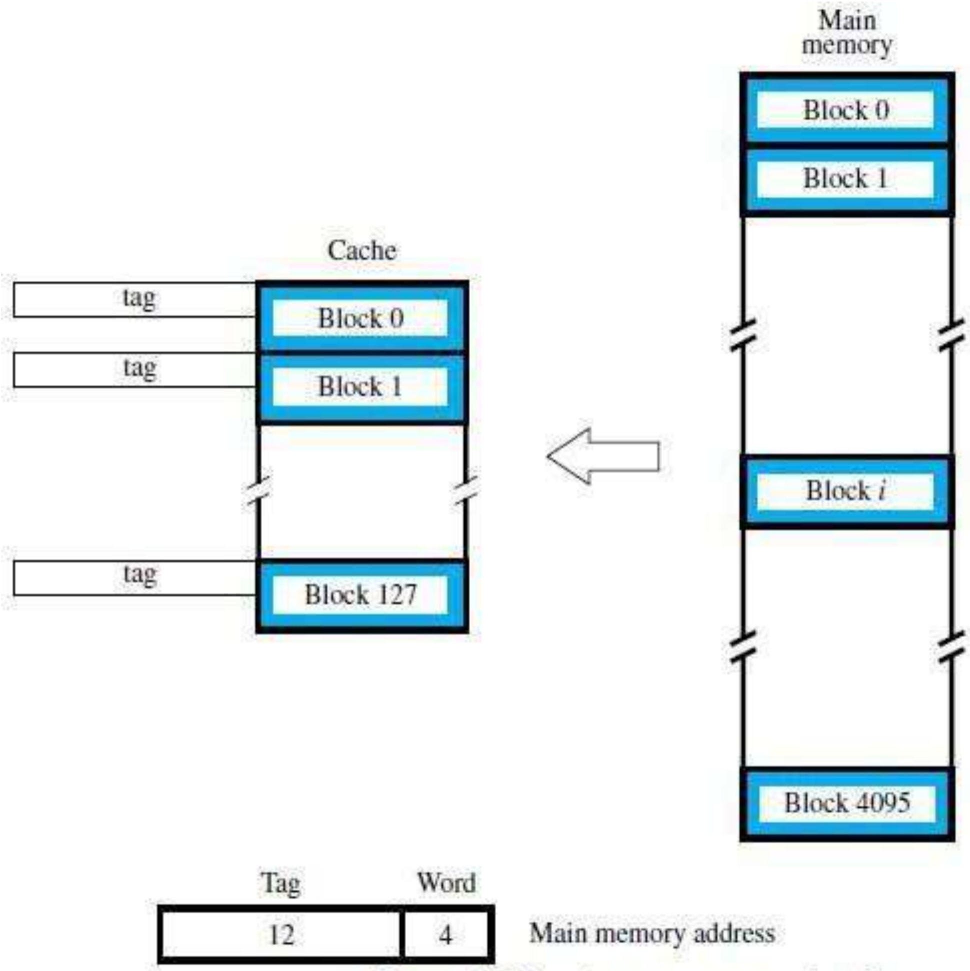


Figure 8.17 Associative-mapped cache.

- It gives complete freedom in choosing the cache-location.
- A new block that has to be brought into the cache has to replace an existing block if the cache is full.
- The memory has to determine whether a given block is in the cache.
- **Advantage:** It is more flexible than direct mapping technique.
- **Disadvantage:** Its cost is high.

SET-ASSOCIATIVE MAPPING

- It is the combination of direct and associative mapping. (Figure 8.18).
- The blocks of the cache are grouped into sets.
- The mapping allows a block of the main-memory to reside in any block of the specified set.
- The cache has 2 blocks per set, so the memory-blocks 0, 64, 128… ...... 4032 maps into cache set „0".
- The cache can occupy either of the two block position within the set.

6 bit set field

- Determines which set of cache contains the desired block.

6 bit tag field

- The tag field of the address is compared to the tags of the two blocks of the set.
- This comparison is done to check if the desired block is present.

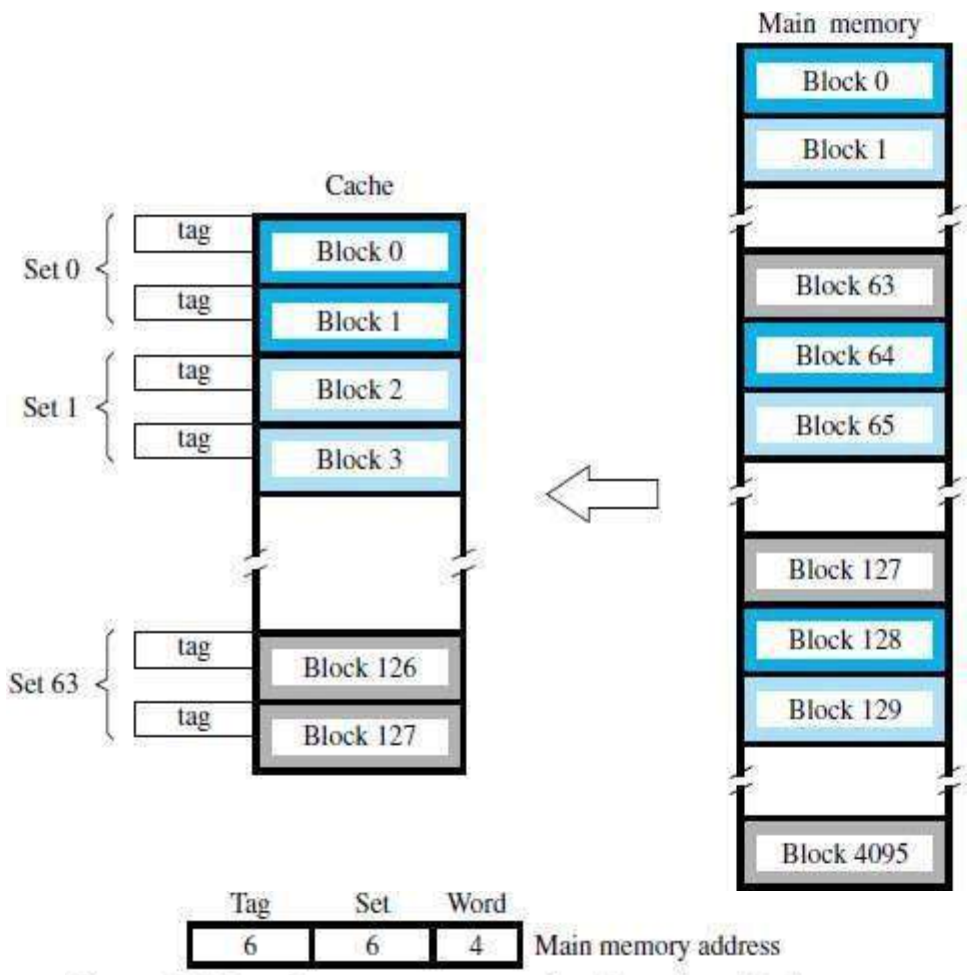


Figure 8.18 Set-associative-mapped cache with two blocks per set.

- The cache which contains 1 block per set is called **direct mapping.** A cache that has „k" blocks per set is called as"**k-way set associative cache**".
- Each block contains a control-bit called a **valid-bit**.
- The Valid-bit indicates that whether the block contains valid-data.
- The dirty bit indicates that whether the block has been modified during its cache residency.

Valid-bit=0 - When power is initially applied to system.
Valid-bit=1 - When the block is loaded from main-memory at first time.

- If the main-memory-block is updated by a source & if the block in the source is already exists in

the cache, then the valid-bit will be cleared to "0‟.

- If Processor & DMA uses the same copies of data then it is called as **Cache Coherence Problem**.

- **Advantages:**

  1) Contention problem of direct mapping is solved by having few choices for block placement.
  2) The hardware cost is decreased by reducing the size of associative search.

REPLACEMENT ALGORITHM

- In direct mapping method, the position of each block is pre-determined and there is no need of replacement strategy.
- In associative & set associative method, the block position is not pre-determined. If the cache is full and if new blocks are brought into the cache, then the cache-controller must decide which of the old blocks has to be replaced.
- When a block is to be overwritten, the block with longest time w/o being referenced is over-written.
- This block is called **Least recently Used (LRU) block** & the technique is called **LRU algorithm**.
- The cache-controller tracks the references to all blocks with the help of block-counter.
- **Advantage:** Performance of LRU is improved by randomness in deciding which block is to be

over- written.

Eg:
Consider 4 blocks/set in set associative cache.

- 2 bit counter can be used for each block.
- When a **'hit'** occurs, then block counter=0; The counter with values originally lower than

the referenced one are incremented by 1 & all others remain unchanged.

- When a **'miss'** occurs & if the set is full, the blocks with the counter value 3 is removed, the newblock is put in its place & its counter is set to "0‟ and other block counters are incremented by 1.

BCS302- DDCO-VTU 2022 scheme

MODULE-5 BASIC PROCESSING UNIT

5.1 Some Fundamental Concepts

To execute an instruction, processor has to perform following 3 steps:

1) Fetch contents of memory-location pointed to by PC. Content of this location is an instruction to be executed. The instructions are loaded into IR, Symbolically, this operation can be written as IR🡐[[PC]]

2) Increment PC by 4 P,🡐 [PC] +4

3) Carry out the actions specified by instruction (in the IR).

The first 2 steps are referred to as **fetch phase**; Step 3 is referred to as **execution phase.**

The operations specified by an instruction can be carried out by performing one or more of the following actions:

1.Read the content of a given memory-location and load them into a register.

2.Read data from one or more register

3.Perform ALU operations and place the result into the register.

4.Store data from a register into a given memory location.

Figure 5.1 shows the single bus organization. ALU and all the registers are interconnected via a single common bus. Data and address line of the external memory bus is connected to the internal processor bus via MDR and MAR respectively(MDR-Memory Data Register and MAR-Memory Address Register).

MDR has 2 inputs and 2 outputs. Data may be loaded

→ into MDR either from memory-bus (external) or

→ from processor-bus (internal).

MAR's input is connected to internal-bus, and MAR"s output is connected to external-bus.Instruction-decoder & control-unit is responsible for

→ issuing the signals that control the operation of all the units inside the processor (and for interacting with memory bus).

→ implementing the actions specified by the instruction (loaded in the IR)

Registers **R0 through R(n-1**) are provided for general purpose use by programmer.

Three registers **Y, Z & TEMP** are used by processor for temporary storage during execution of some instructions.

These are transparent to the programmer i.e. programmer need not be concerned with them because they are never referenced explicitly by any instruction.

MUX(Multiplexer) selects either

→ output of Y or

→ constant-value 4(is used to increment PC content).This is provided as **input A of ALU.**

B input of ALU is obtained directly from processor-bus.

As instruction execution progresses, data are transferred from one register to another, often passing through **ALU to perform arithmetic or logic operation**.

An instruction can be executed by performing one or more of the following operations:

1) **Transfer a word of data from one processor-register** to another or to the ALU.

2) **Perform arithmetic or a logic operation** and store the result in a processor-register.

3) Fetch the contents of a given memory-location and **load them into a processor-register**.

4) **Store a word of data** from a processor-register **into a given memory-location**.

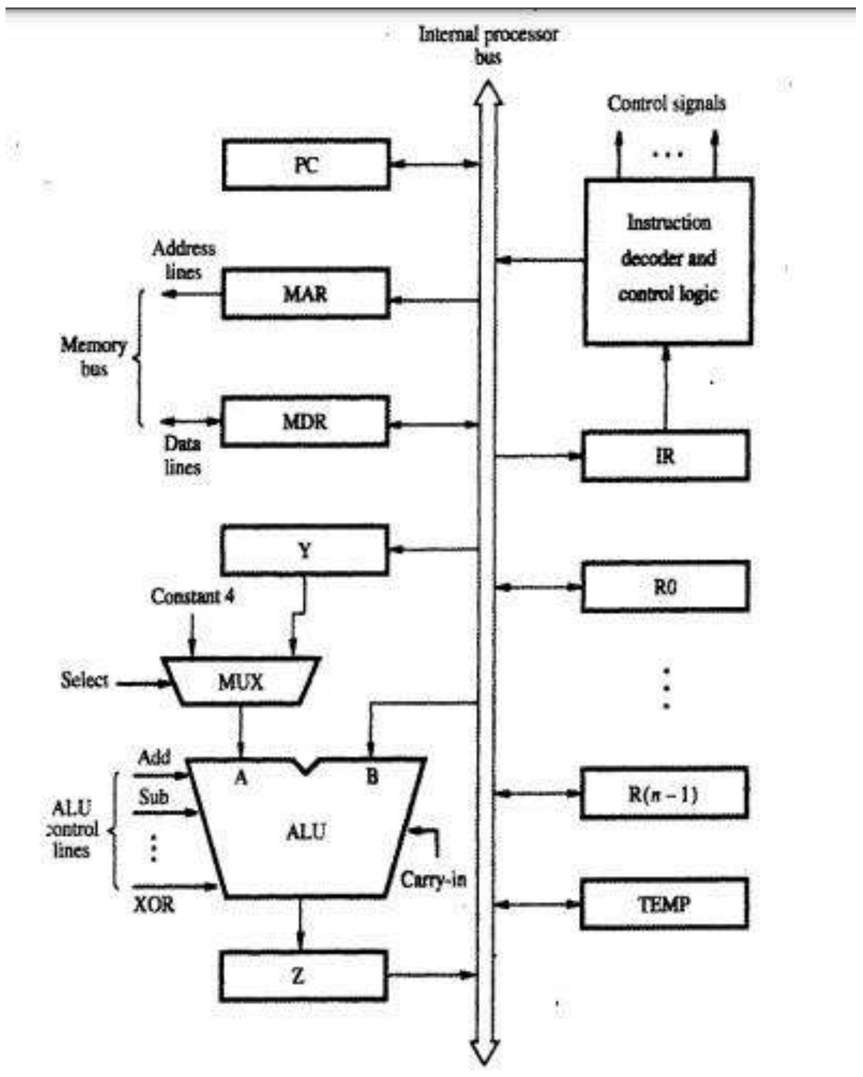


Figure 5.1: Single bus organization of the data path inside a processor

Disadvantage: Only one data word can be transferred over the bus in a clock cycle.

Solution: Providing multiple data-paths allows several data transfer to take place in parallel.

5.1.1 Register Transfers

Instruction execution involves a sequence of steps in which data are transferred from one register to another.

Input & output of register R_i is connected to bus via switches controlled by 2 control-signals: Ri_{in} & Ri_{out}. These are called **gating signals.**

When Ri_{in}=1, data on bus is loaded into $R_{i.}$ Similarly, when Ri_{out}=1, content of R_i is placed on bus. When Ri_{out}=0, bus can be used for transferring data from other registers.

For example, **MOVE R1,R2**

This transfers the content of register R1 to R2. This can be accomplished as follows.

1. Enable the output of Register R1 by setting $R1_{out}$ to 1.

2. Enable the input of Register R2 by setting $R2_{in}$ to 1.

All operations and data transfers within the processor take place within time-periods defined by the **processor clock.** When edge-triggered flip-flops are not used, 2 or more clock-signals may be needed to guarantee proper transfer of data. This is known as **multiphase clocking**.

Input & Output Gating for one Register Bit

A 2-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When Ri_{in}=1, mux selects data on bus. This data will be loaded into flip-flop at rising-edge of clock. When Ri_{in}=0, mux feeds back the value currently stored in flip-flop. Q output of flip-flop is connected to bus via a tri-state gate. When Ri_{out}=0, gate's output is in the high-impedance state. (This corresponds to the open circuit state of a switch). When Ri_{out}=1, the gate drives the bus to 0 or 1, depending on the value of Q.

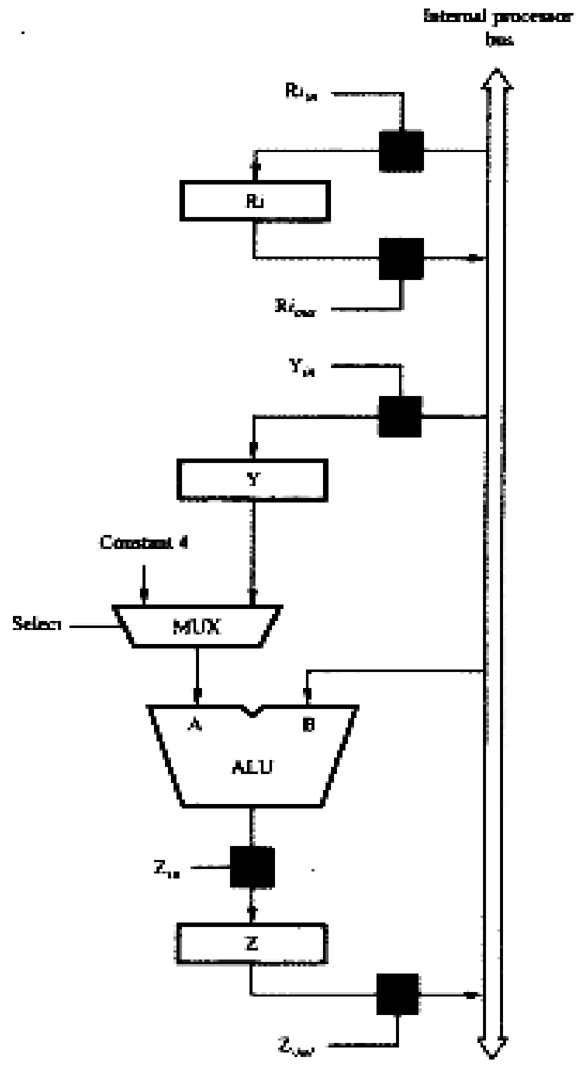


Figure 5.2: Input and output gating for a register

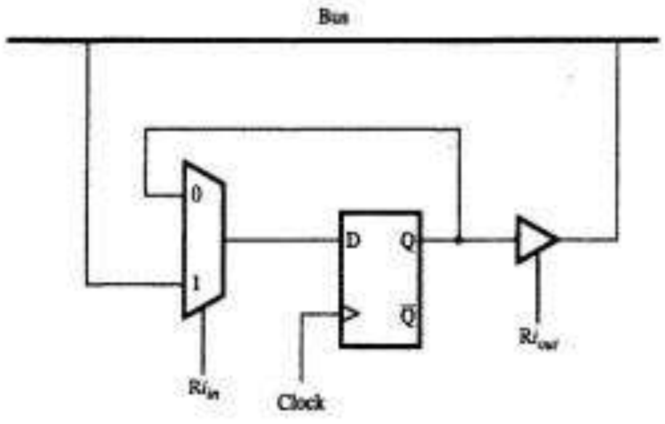


Figure 5.3: Input and output gating for one-bit register

5.1.2 Performing ALU operations

The ALU performs arithmetic operations on the 2 operands applied to its A and B inputs. One of the operands is output of MUX & the other operand is obtained directly from bus. The result (produced by the ALU) is stored temporarily in register Z.

Eg: Add R1,R2,R3

The sequence of operations for [R3]$\leftarrow$[R1]+[R2] is as follows

1) **$R1_{out}$, Y_{in}** //transfer the contents of R1 to Y register

2) **$R2_{out}$, Select Y, Add, Z_{in}** //R2 contents are transferred directly to B input of ALU.

// The numbers of added. Sum stored in register Z

3) **Z_{out}, $R3_{in}$** //sum is transferred to register R3

The signals are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive.

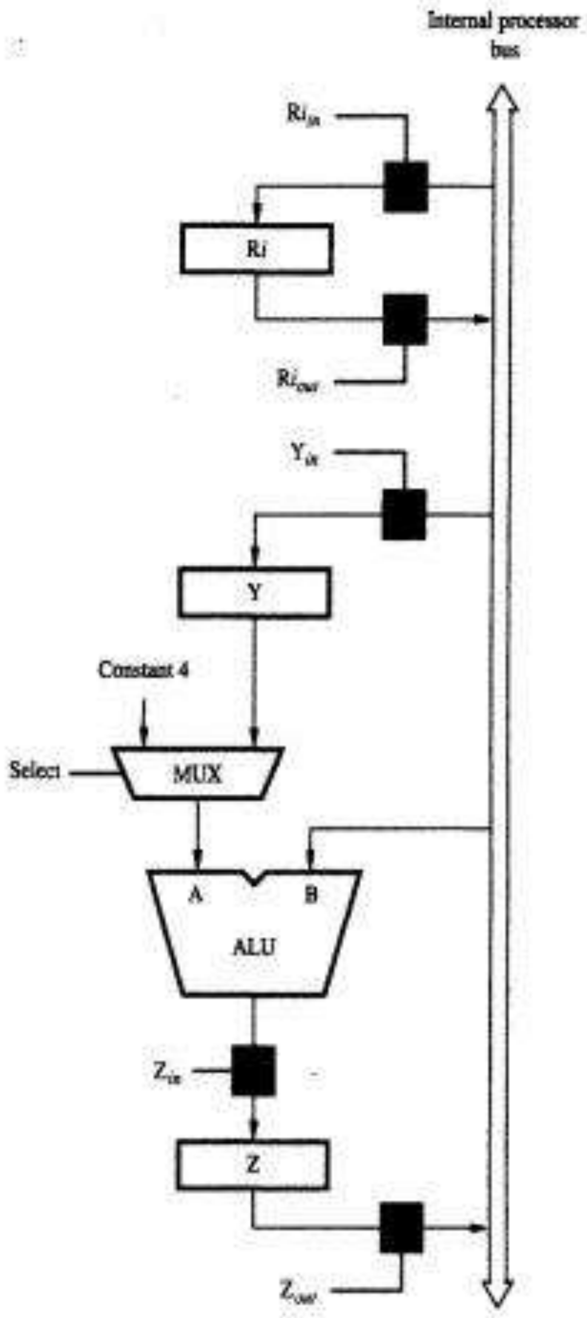


Figure 5.4: ALU operation

(Note: In this Figure 5.4 , replace Register Ri with Registers R1, R2, R3)

5.1.3 Fetching a word from Memory

To fetch instruction/data from memory, processor transfers required address to MAR (whose output is connected to address-lines of memory-bus). At the same time, processor issues Read signal on control-lines of memory-bus. When requested-data are received from memory, they are stored in MDR. From MDR, they are transferred to other registers.

The response time of each memory access varies. For this MFC (Memory Function Completed): is used. It is the signal sent from Addressed-device to the processor. MFC informs the processor that the requested operation is completed by addressed device.

Thus MFC is set to 1 to indicate that the contents of the specified location

→ have been read &

→ are available on data-lines of memory-bus

Consider the instruction **Move (R1),R2.** The sequence of steps is:

1) **R1$_{out}$, MAR$_{in}$, Read** ;desired address is loaded into MAR & Read command is issued
2) **MDR$_{inE}$, WMFC** ;load MDR from memory bus & Wait for MFC response from memory.
3) **MDR$_{out}$, R2$_{in}$** ;load R2 from MDR where WMFC=control signal that causes processor's control circuitry to wait for arrival of MFC signal

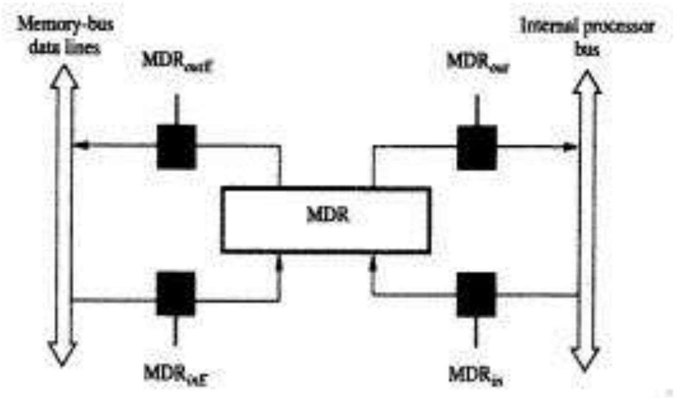


Figure 5.5:Connection and control signal for Register MDR

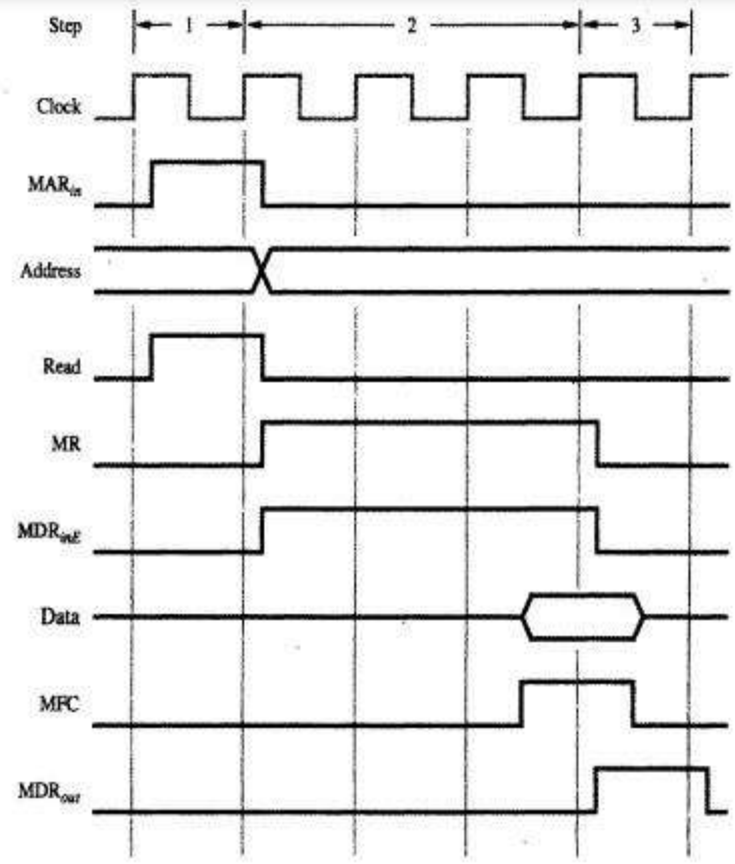


Figure 5.6: Timing of a memory Read operation

5.1.4 Storing a word in memory

Consider the instruction **Move R2,(R1).** This requires the following sequence:

1) **R1out, MARin** ;desired address is loaded into MAR

2) **R2out, MDRin, Write** ;data to be written are loaded into MDR & Write command is issued

3) **MDRoutE, WMFC** ;load data into memory location pointed by R1 from MDR

5.2. Execution of a Complete Instruction.

Consider the instruction **Add (R3),R1** which adds the contents of a memory-location pointed by R3 to register R1. **Executing this instruction requires the following actions:**

1) Fetch the instruction.

2) Fetch the first operand.

3) Perform the addition.

4) Load the result into R1.

Control sequence for execution of this instruction is as follows

1) PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}

2) Z_{out}, PC_{in}, Y_{in}, WMFC

3) MDR_{out}, IR_{in}

4) $R3_{out}$, MAR_{in}, Read

5) $R1_{out}$, Y_{in}, WMFC

6) MDR_{out}, SelectY, Add, Z_{in}

7) Z_{out}, $R1_{in}$, End

Instruction execution proceeds as follows:

Step1→ The instruction-fetch operation is initiated by loading contents of PC into MAR & sending a Read request to memory. The Select signal is set to Select4, which causes the Mux to select constant 4. This value is added to operand at input B (PC"s content), and the result is stored in Z

Step2→ Updated value in Z is moved to PC.

Step3→ Fetched instruction is moved into MDR and then to IR.

Step4→ Contents of R3 are loaded into MAR & a memory read signal is issued.

Step5→ Contents of R1 are transferred to Y to prepare for addition.

Step6→ When Read operation is completed, memory-operand is available in MDR, and the

addition is performed.

Step7→ Sum is stored in Z, then transferred to R1.The End signal causes a new instruction fetch cycle to begin by returning to step1.

BRANCHING INSTRUCTIONS

Control sequence for an unconditional branch instruction is as follows:

1) PCout, MARin, Read, Select4, Add, Zin

2) Zout, PCin, Yin, WMFC

3) MDRout, IRin

4) Offset-field-of-IRout, Add, Zin

5) Zout, PCin, End

The processing starts, as usual, the fetch phase ends in step3.

In step 4, the offset-value is extracted from IR by instruction-decoding circuit. Since the updated value of PC is already available in register Y, the offset X is gated onto the bus, and an addition operation is performed.

In step 5, the result, which is the branch-address, is loaded into the PC. The offset X used in a branch instruction is usually the difference between the branch target-address and the address immediately following the branch instruction. (For example, if the branch instruction is at location 1000 and branch target-address is 1200, then the value of X must be 196, since the PC will be containing the address 1004 after fetching the instruction at location 1000).

In case of conditional branch, we need to check the status of the condition-codes before loading a new value into the PC.

e.g.: Offset-field-of-IRout, Add, Zin,

If N=0 then End If N=0, processor returns to step 1 immediately after step 4.

If N=1, step 5 is performed to load a new value into PC

5.3 Pipelining:

The speed of execution of programs is influenced by many factors.

1. One way to improve performance is to use faster circuit technology to implement the processor and the main memory.

2. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased, even though the time needed to perform any one operation is not changed.

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other.

Let Fi and Ei refer to the fetch and execute steps for instruction Ii. Execution of a program consists of a sequence of fetch and execute steps, as shown in **Figure 5.7**

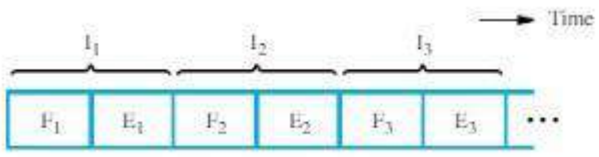


Figure 5.7 Sequential execution

.Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in **Figure 5.8.**

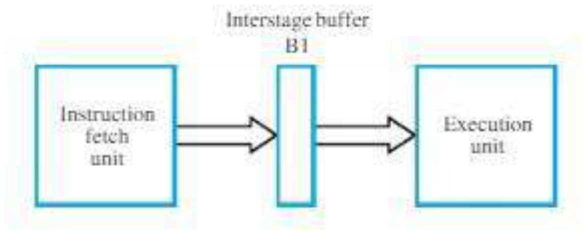


Figure 5.8 Hardware organization

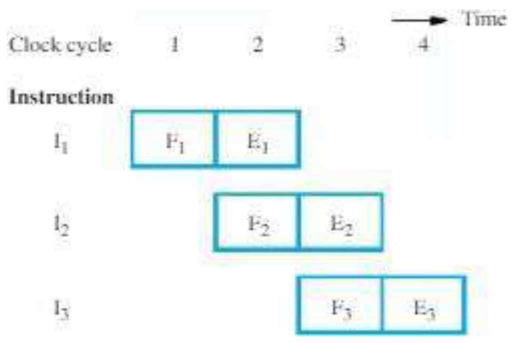


Figure 5.9: Pipelined execution (2 stage)

The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction.

Operation of the computer proceeds as in Figure 5.9. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2). Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1).

By the end of second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1,replacing I1,which is no longer needed. StepE2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit.

In this manner, both the fetch and execute units are kept busy all the time.

In summary, the fetch and execute units in Figure 5.3 constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.

The sequence of events for this case is shown in **Figure 5.10.**

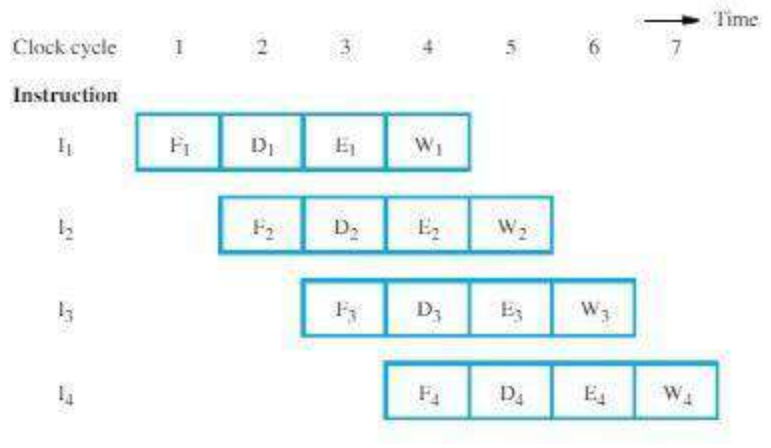


Figure 5.10: Pipelined execution (4 stage)

Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in **Figure 5.11**

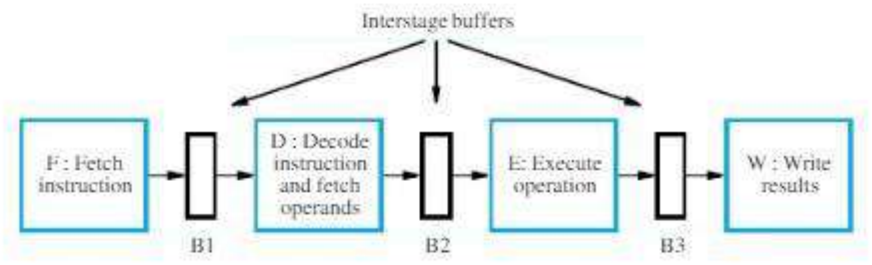


Figure 5.11 Hardware organization

These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

•**Buffer B1** holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.

•**Buffer B2** holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle3.Thebuffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.

•**Buffer B3** holds the results produced by the execution unit and the destination information for instruction I1.

5.3.1 Role of cache:

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage.

If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time.

In **Figure 5.12,** the clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

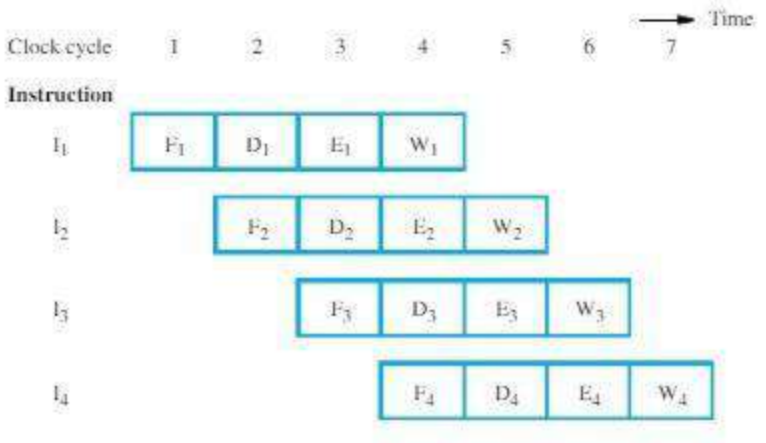

Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4

Figure 5.12: Instruction execution (4 stage pipeline)

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor.

This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

5.3.2 Pipeline performance

The pipelined processor in Figure 5.6 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.

Let us consider an example of, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted as in **Figure 5.13.**

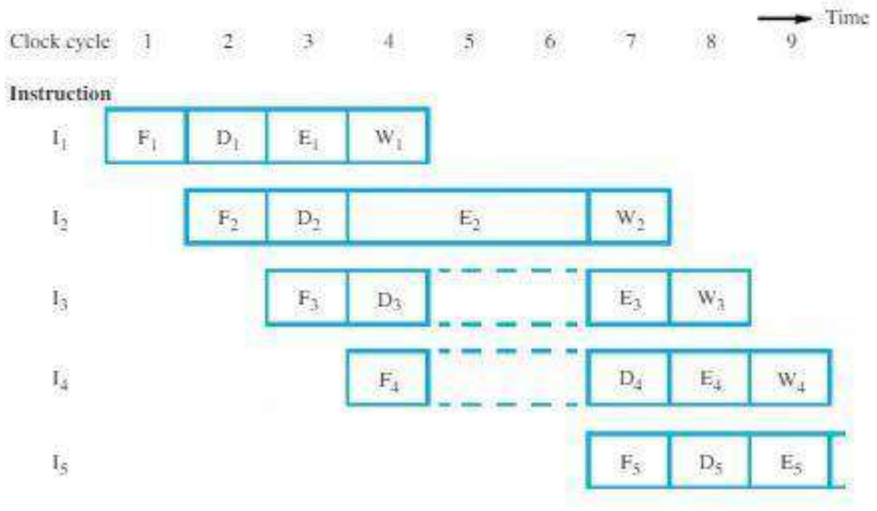

Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
I_5 F_5 D_5 E_5

Figure 5.13: Execution unit takes more than one cycle for execution

Here instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed.

Pipelined operation in **Figure 5.13** is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called **a hazard**.

There are three types of Hazards:

1. Data hazard

2. Instruction or control hazard

3. Structural hazard

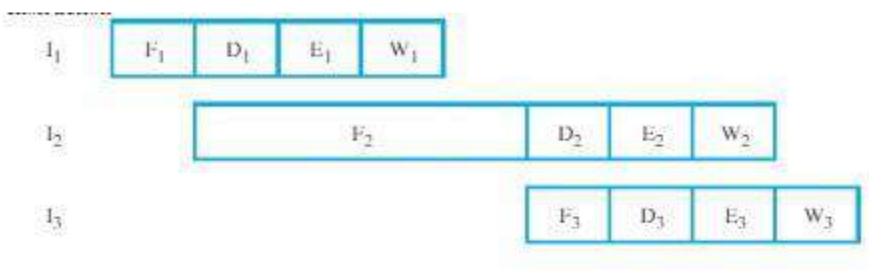
Data hazard

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

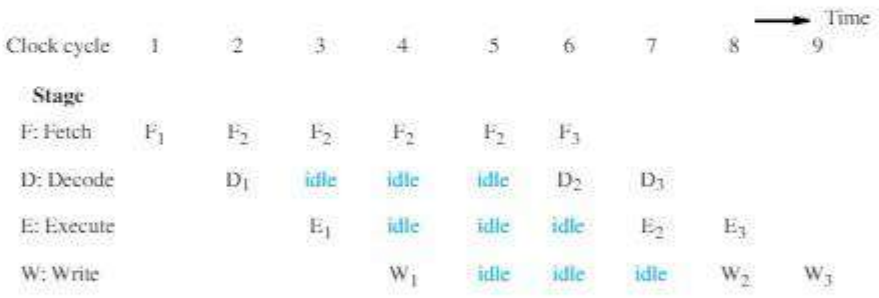
Control hazards or instruction hazards

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called control hazards or instruction hazards. **Figure 5.14** has instruction hazard with it.

Instruction I1 is fetched from the cache in cycle1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2,results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles



Clock cycle	1	2	3	4	5	6	7	8	9 → Time
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

Figure 5.14 Instruction Hazard

Structural hazard

A third type of hazard that may be encountered in pipelined operation is known as a structural hazard. This is the situation when two instructions require the use of a given hardware resource at the same time.

Example: Load X(R1),R2

The memory address, X+[R1], is computed in step E2 in cycle4, then memory access takes place in cycle5.The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6 which is shown in **Figure 5.15.**

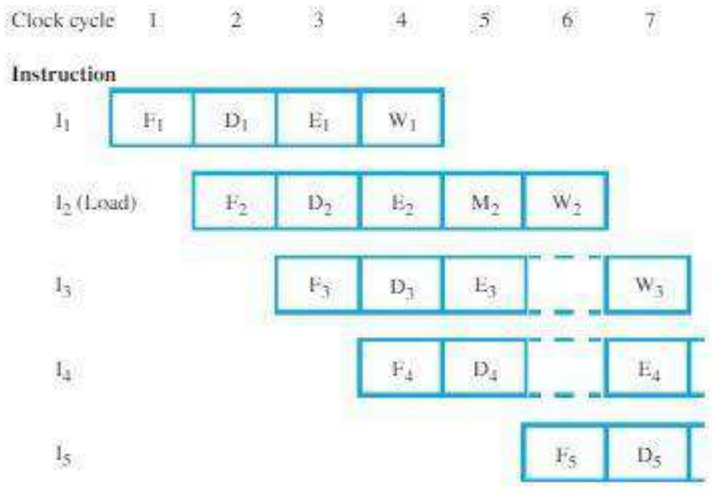


Figure 5.15: Structural hazard

Even though the instructions and their data are all available, the pipeline stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed.

Many processors use separate instruction and data caches to avoid this delay.

An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.