

UNIT I
COMBINATIONAL LOGIC

Combinational Circuits – Karnaugh Map - Analysis and Design Procedures – Binary Adder –
Subtractor – Decimal Adder - Magnitude Comparator – Decoder – Encoder – Multiplexers -
Demultiplexers

INTRODUCTION:

The digital system consists of two types of circuits, namely

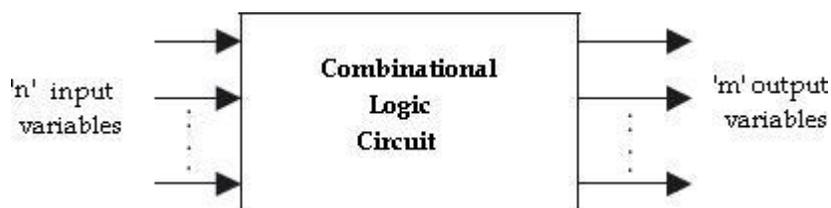
- (i) Combinational circuits
- (ii) Sequential circuits

Combinational circuit consists of logic gates whose output at any time is determined from the present combination of inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

Sequential logic circuit comprises both logic gates and the state of storage elements such as flip-flops. As a consequence, the output of a sequential circuit depends not only on present value of inputs but also on the past state of inputs.

In the previous chapter, we have discussed binary numbers, codes, Boolean algebra and simplification of Boolean function and logic gates. In this chapter, formulation and analysis of various systematic designs of combinational circuits will be discussed.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from inputs and output signals are generated according to the logic circuits employed in it. Binary information from the given data transforms to desired output data in this process. Both input and output are obviously the binary signals, *i.e.*, both the input and output signals are of two possible states, logic 1 and logic 0.



Block diagram of a combinational logic circuit

For n number of input variables to a combinational circuit, 2^n possible combinations of binary input states are possible. For each possible combination, there is one and only one possible output combination. A combinational logic circuit can be described by m Boolean functions and each output can be expressed in terms of n input variables.

DESIGN PROCEDURE:

Any combinational circuit can be designed by the following steps of design procedure.

1. The problem is stated.
2. Identify the input and output variables.
3. The input and output variables are assigned letter symbols.
4. Construction of a truth table to meet input -output requirements.
5. Writing Boolean expressions for various output variables in terms of input variables.
6. The simplified Boolean expression is obtained by any method of minimization – algebraic method, Karnaugh map method, or tabulation method.
7. A logic diagram is realized from the simplified boolean expression using logic gates.

The following guidelines should be followed while choosing the preferred form for hardware implementation:

1. The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.
2. There should be a minimum number of interconnections.
3. Limitation on the driving capability of the gates should not be ignored.

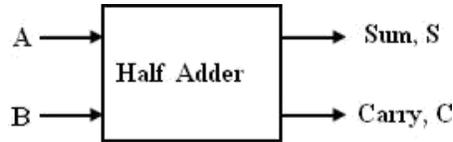
ARITHMETIC CIRCUITS – BASIC BUILDING BLOCKS:

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction.

The basic building blocks that form the basis of all hardware used to perform the arithmetic operations on binary numbers are half-adder, full adder, half-subtractor, full-subtractor.

Half-Adder:

A half-adder is a combinational circuit that can be used to add two binary bits. It has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.



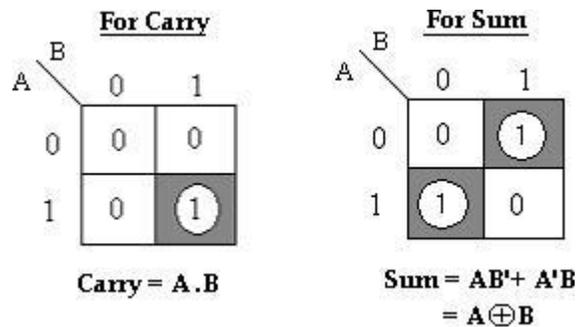
Block schematic of half-adder

The truth table of a half-adder, showing all possible input combinations and the corresponding outputs are shown below.

Inputs		Outputs	
A	B	Carry (C)	Sum (S)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table of half-adder

K-map simplification for carry and sum:



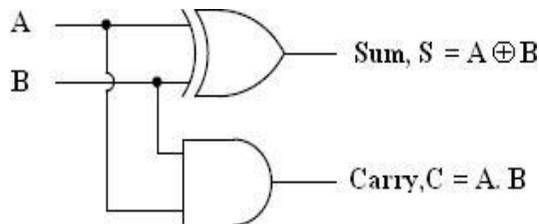
The Boolean expressions for the SUM and CARRY outputs are given by the equations,

$$\text{Sum, } S = A'B + AB' = A \oplus B$$

$$\text{Carry, } C = A \cdot B$$

The first one representing the SUM output is that of an EX-OR gate, the second one representing the CARRY output is that of an AND gate.

The logic diagram of the half adder is,

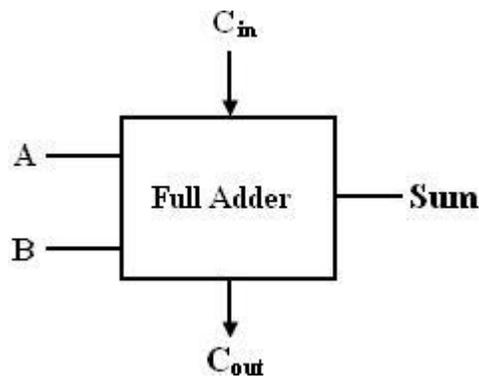


Logic Implementation of Half-adder

Full-Adder:

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of 3 inputs and 2 outputs.

Two of the input variables, represent the significant bits to be added. The third input represents the carry from previous lower significant position. The block diagram of full adder is given by,



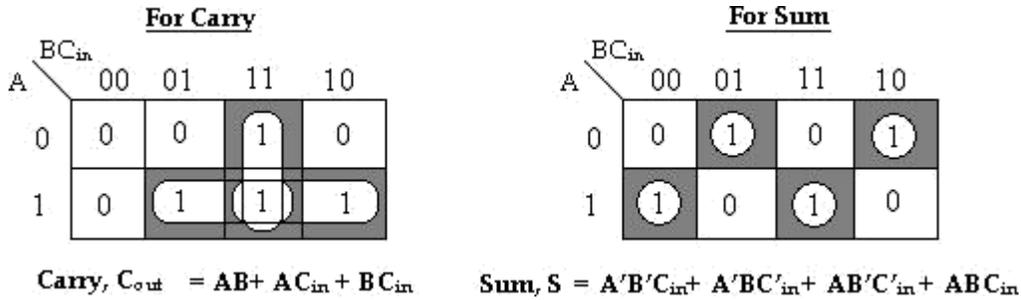
Block schematic of full-adder

The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. As there are three input variables, eight different input combinations are possible. The truth table is shown below,

Truth Table:

Inputs			Outputs	
A	B	C _{in}	Sum (S)	Carry (C _{out})
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

To derive the simplified Boolean expression from the truth table, the Karnaugh map method is adopted as,

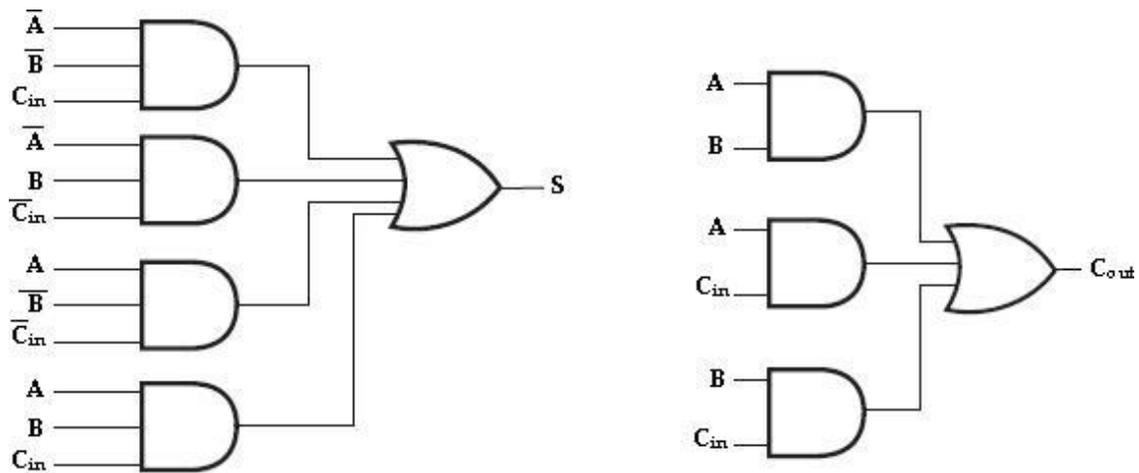


The Boolean expressions for the SUM and CARRY outputs are given by the equations,

$$\text{Sum, } S = A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in}$$

$$\text{Carry, } C_{out} = AB + AC_{in} + BC_{in}.$$

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Products

The logic diagram of the full adder can also be implemented with two half-adders and one OR gate. The S output from the second half adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

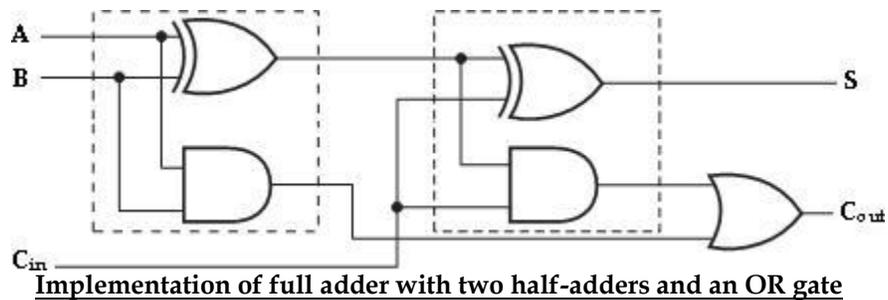
$$\begin{aligned} \text{Sum} &= C_{in} \uparrow (A \uparrow B) \\ &= C_{in} \uparrow (A'B + AB') \\ &= C'_{in} (A'B + AB') + C_{in} (A'B + AB') \\ &= C'_{in} (A'B + AB') + C_{in} (AB + A'B') \\ &= A'BC'_{in} + AB'C'_{in} + ABC_{in} + A'B'C_{in}. \end{aligned}$$

$$[x \uparrow y = x'y + xy']$$

$$[(x'y + xy')]' = (xy + x'y')$$

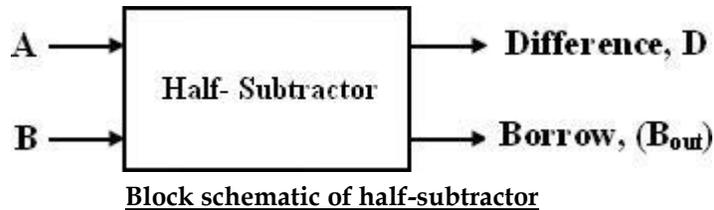
and the carry output is,

$$\begin{aligned}
 \text{Carry, } C_{out} &= AB + C_{in} (A'B + AB') \\
 &= AB + A'BC_{in} + AB'C_{in} \\
 &= AB (C_{in}+1) + A'BC_{in} + AB'C_{in} && [C_{in}+1=1] \\
 &= ABC_{in} + AB + A'BC_{in} + AB'C_{in} \\
 &= AB + AC_{in} (B+B') + A'BC_{in} \\
 &= AB + AC_{in} + A'BC_{in} \\
 &= AB (C_{in}+1) + AC_{in} + A'BC_{in} && [C_{in}+1=1] \\
 &= ABC_{in} + AB + AC_{in} + A'BC_{in} \\
 &= AB + AC_{in} + BC_{in} (A + A') \\
 &= AB + AC_{in} + BC_{in}.
 \end{aligned}$$



Half -Subtractor:

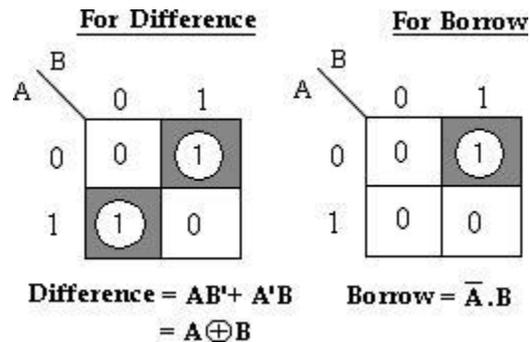
A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a '1' has been borrowed to perform the subtraction.



The truth table of half-subtractor, showing all possible input combinations and the corresponding outputs are shown below.

Input		Output	
A	B	Difference (D)	Borrow (Bout)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map simplification for half subtractor:



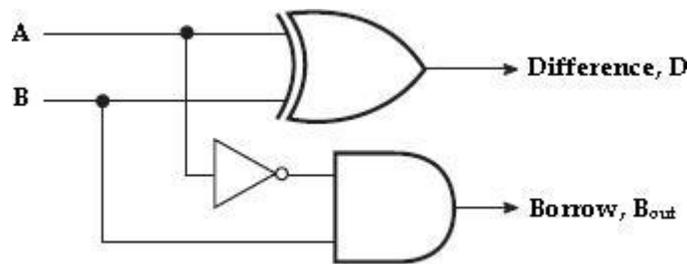
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = $A'B + AB'$ = $A \oplus B$

Borrow, B_{out} = $A' \cdot B$

The first one representing the DIFFERENCE (D) output is that of an exclusive-OR gate, the expression for the BORROW output (B_{out}) is that of an AND gate with input A complemented before it is fed to the gate.

The logic diagram of the half subtractor is,



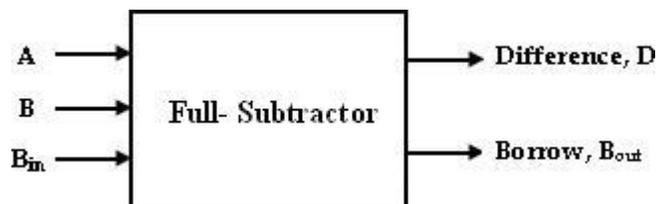
Logic Implementation of Half-Subtractor

Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, i.e., the minuend is complemented, an AND gate can be used to implement the BORROW output.

Full Subtractor:

A full subtractor performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a '1' has already been borrowed by the previous adjacent lower minuend bit or not.

As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o. The



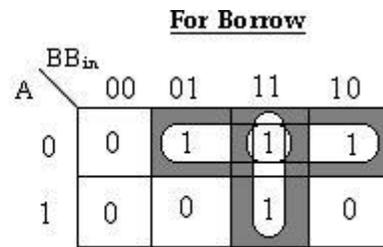
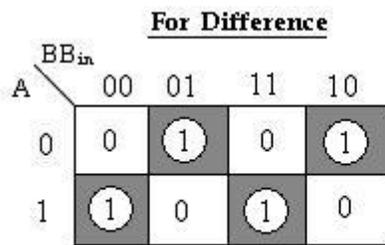
BORROW output bit tells whether the minuend bit needs to borrow a '1' from the next possible higher minuend bit.

Block schematic of full-addder

The truth table for full-subtractor is,

Inputs			Outputs	
A	B	B _{in}	Difference(D)	Borrow(B _{out})
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K-map simplification for full-subtractor:



Difference, D = $A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

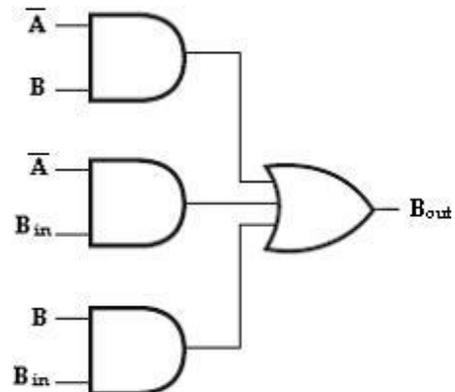
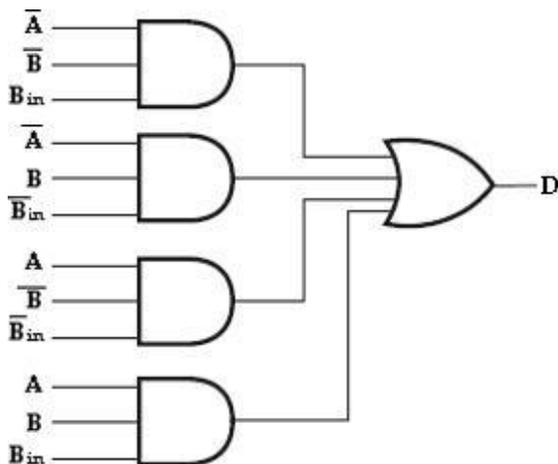
Borrow, B_{out} = $A'B + A'B_{in} + BB_{in}$

The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = $A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

Borrow, B_{out} = $A'B + A'B_{in} + BB_{in}$.

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Products

The logic diagram of the full-subtractor can also be implemented with two half-subtractors and one OR gate. The difference, D output from the second half subtractor is the exclusive-OR of B_{in} and the output of the first half-subtractor, giving

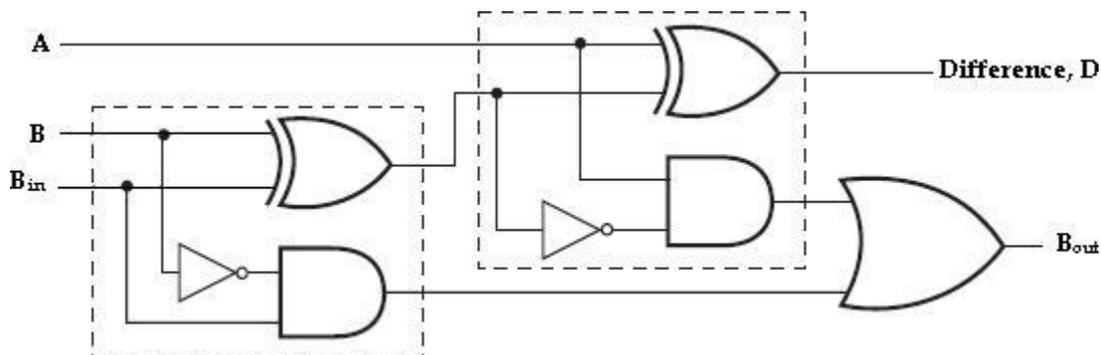
$$\begin{aligned}
 \text{Difference, } D &= B_{in} \oplus (A \oplus B) & [x \oplus y &= x'y + xy'] \\
 &= B_{in} \oplus (A'B + AB') \\
 &= B'_{in} (A'B + AB') + B_{in} (A'B + AB') & [(x'y + xy')' &= (xy + x'y')] \\
 &= B'_{in} (A'B + AB') + B_{in} (AB + A'B') \\
 &= A'B B'_{in} + AB' B'_{in} + A B B_{in} + A'B B_{in}
 \end{aligned}$$

and the borrow output is,

$$\begin{aligned}
 \text{Borrow, } B_{out} &= A'B + B_{in} (A'B + AB') & [(x'y + xy')' &= (xy + x'y')] \\
 &= A'B + B_{in} (AB + A'B') \\
 &= A'B + A B B_{in} + A'B' B_{in} \\
 &= A'B (B_{in} + 1) + A B B_{in} + A'B' B_{in} & [C_{in} + 1 &= 1] \\
 &= A'B B_{in} + A'B + A B B_{in} + A'B' B_{in} \\
 &= A'B + B B_{in} (A + A') + A'B' B_{in} & [A + A' &= 1] \\
 &= A'B + B B_{in} + A'B' B_{in} \\
 &= A'B (B_{in} + 1) + B B_{in} + A'B' B_{in} & [C_{in} + 1 &= 1] \\
 &= A'B B_{in} + A'B + B B_{in} + A'B' B_{in} \\
 &= A'B + B B_{in} + A'B_{in} (B + B') \\
 &= A'B + B B_{in} + A'B_{in}
 \end{aligned}$$

Therefore,

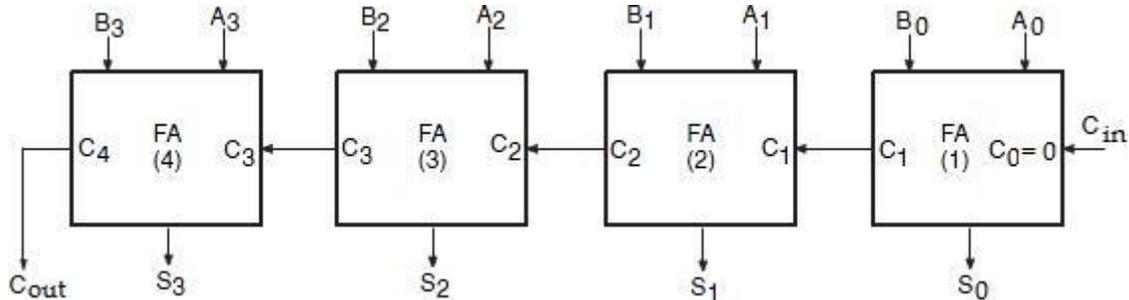
we can implement full-subtractor using two half-subtractors and OR gate as,



Implementation of full-subtractor with two half-subtractors and an OR gate

Binary Adder (Parallel Adder):

The 4-bit binary adder using full adder circuits is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in figure below.



4-bit binary parallel Adder

Since all the bits of augend and addend are fed into the adder circuits simultaneously and the additions in each position are taking place at the same time, this circuit is known as parallel adder.

Let the 4-bit words to be added be represented by,

$A_3A_2A_1A_0 = 1111$ and $B_3B_2B_1B_0 = 0011$.

Significant place	4	3	2	1	
Input carry	1	1	1	0	
Augend word A :	1	1	1	1	
Addend word B :	0	0	1	1	
	1	0	0	1	0 ← Sum
	↑				
Output Carry					

The bits are added with full adders, starting from the least significant position, to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The carry output of the lower order stage is connected to the carry input of the next higher order stage. Hence this type of adder is called ripple-carry adder.

In the least significant stage, A_0 , B_0 and C_0 (which is 0) are added resulting in sum S_0 and carry C_1 . This carry C_1 becomes the carry input to the second stage. Similarly in the second stage, A_1 , B_1 and C_1 are added resulting in sum S_1 and carry C_2 , in the third stage, A_2 , B_2 and C_2 are added resulting in sum S_2 and carry C_3 , in the fourth stage, A_3 , B_3 and C_3 are added resulting in sum S_3 and C_4 , which is the output carry. Thus the circuit results in a sum ($S_3S_2S_1S_0$) and a carry output (C_{out}).

Though the parallel binary adder is said to generate its output immediately after the inputs are applied, its speed of operation is limited by the carry propagation delay

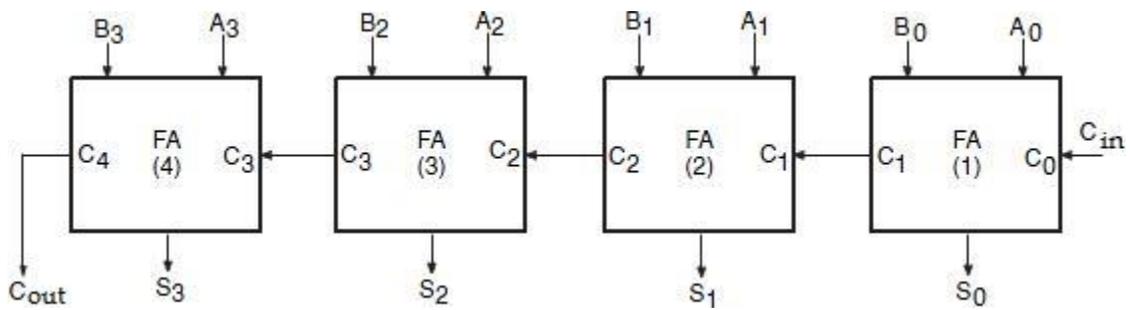
through all stages. However, there are several methods to reduce this delay.

One of the methods of speeding up this process is look-ahead carry addition which eliminates the ripple-carry delay.

Carry Propagation–Look-Ahead Carry Generator:

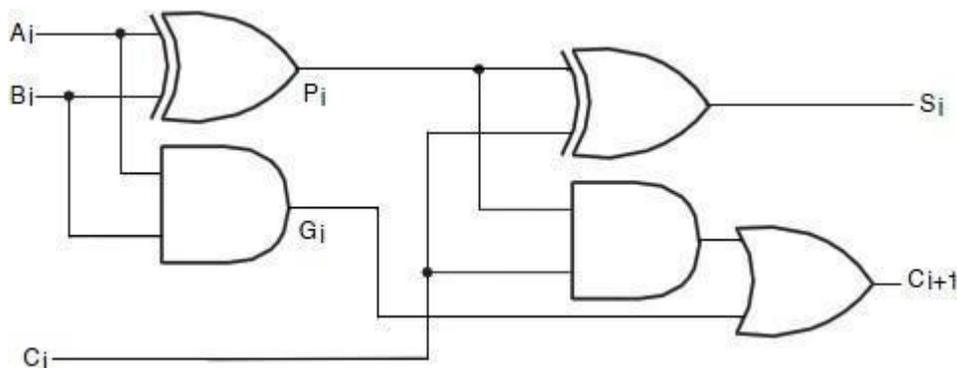
In Parallel adder, all the bits of the augend and the addend are available for computation at the same time. The carry output of each full-adder stage is connected to the carry input of the next high-order stage. Since each bit of the sum output depends on the value of the input carry, time delay occurs in the addition process. This time delay is called as **carry propagation delay**.

For example, addition of two numbers (0011+ 0101) gives the result as 1000. Addition of the LSB position produces a carry into the second position. This carry when added to the bits of the second position, produces a carry into the third position. This carry when added to bits of the third position, produces a carry into the last position. The sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous position. i.e., the adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example, if each full adder is considered to have a propagation delay of 30nsec, then S_3 will not react its correct value until 90 nsec after LSB is generated. Therefore total time required to perform addition is $90 + 30 = 120$ nsec.



4-bit Parallel Adder

The method of speeding up this process by eliminating inter stage carry delay is called **look ahead-carry addition**. This method utilizes logic gates to look at the lower order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions: carry generate and carry propagate.



Full-Adder circuit

Consider the circuit of the full-adder shown above. Here we define two functions: carry generate (G_i) and carry propagate (P_i) as,

$$\text{Carry generate, } G_i = A_i \uparrow B_i$$

$$\text{Carry propagate, } P_i = A_i \uparrow B_i$$

the output sum and carry can be expressed as,

$$S_i = P_i \uparrow C_i$$

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

G_i (carry generate), it produces a carry 1 when both A_i and B_i are 1, regardless of the input carry C_i .

P_i (carry propagate) because it is the term associated with the propagation of the carry from C_i to C_{i+1} .

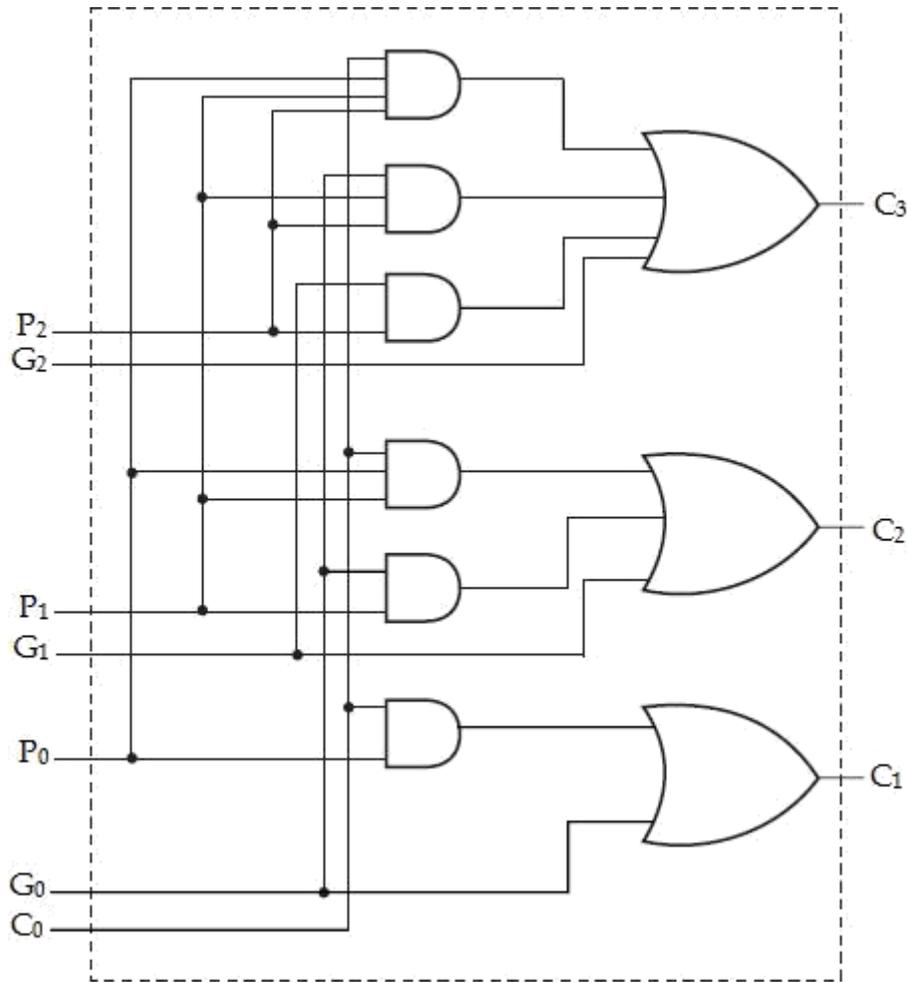
The Boolean functions for the carry outputs of each stage and substitute for each C_i its value from the previous equation:

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$

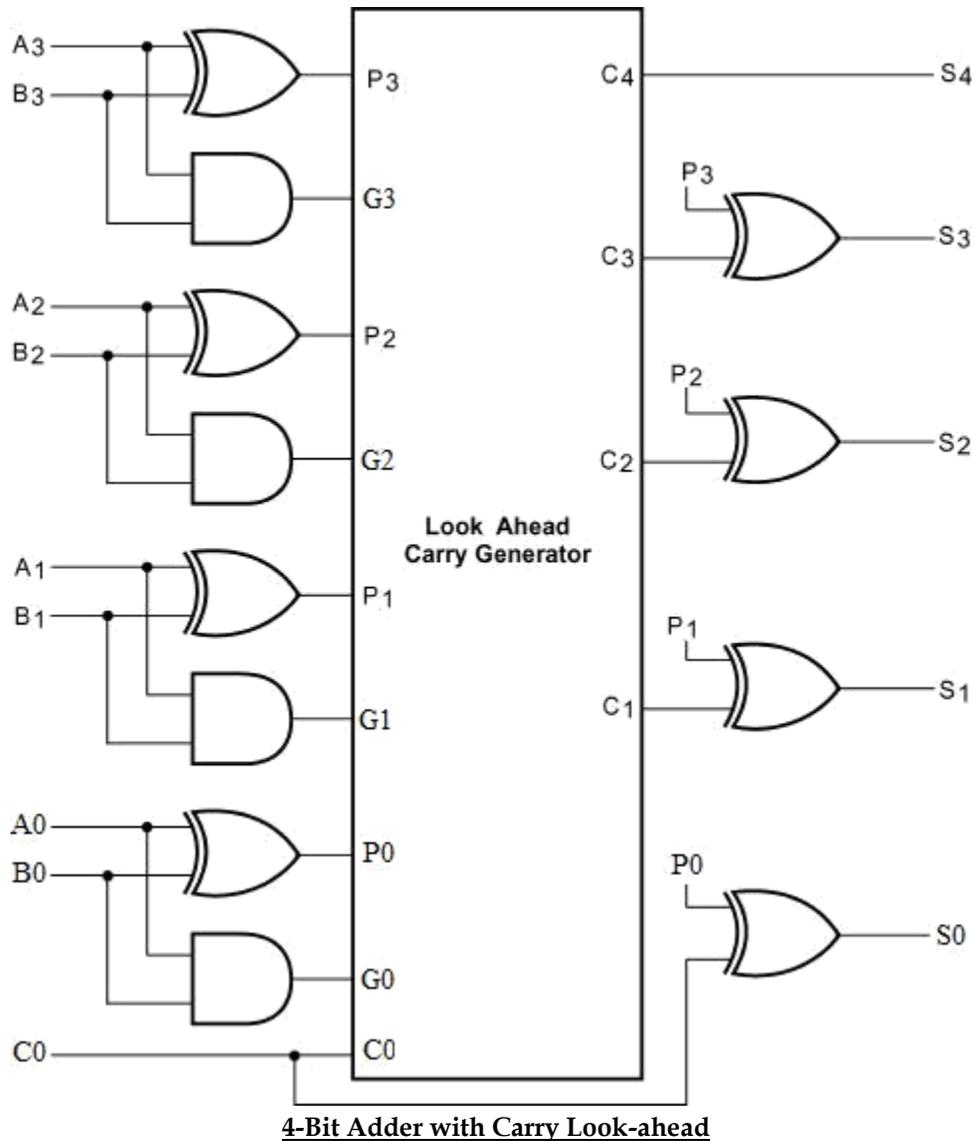
$$\begin{aligned} C_3 &= G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0) \\ &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{aligned}$$



Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate. The three Boolean functions for C_1 , C_2 and C_3 are implemented in the carry look-ahead generator as shown below. Note that C_3 does not have to wait for C_2 and C_1 to propagate; in fact C_3 is propagated at the same time as C_1 and C_2 .

Logic diagram of Carry Look-ahead Generator

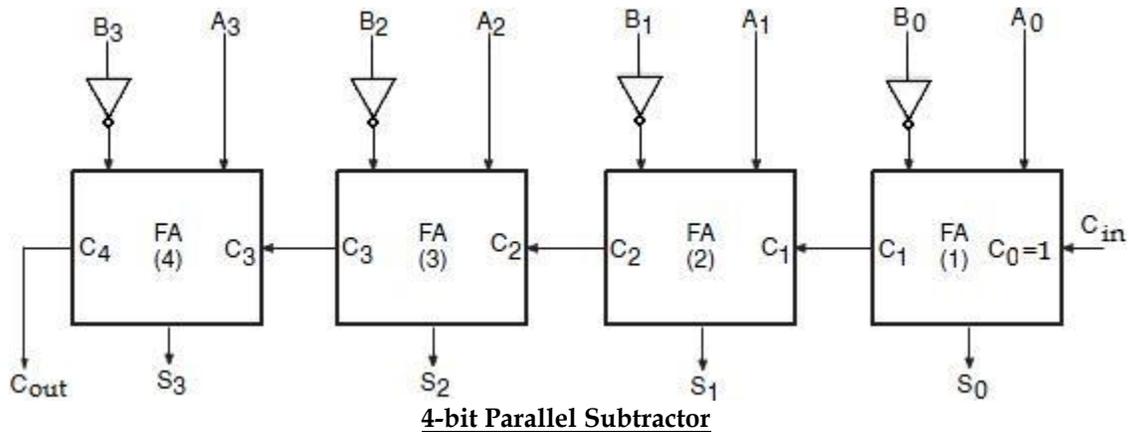
Using a Look-ahead Generator we can easily construct a 4-bit parallel adder with a Look-ahead carry scheme. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry look-ahead generator and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.



Binary Subtractor (Parallel Subtractor):

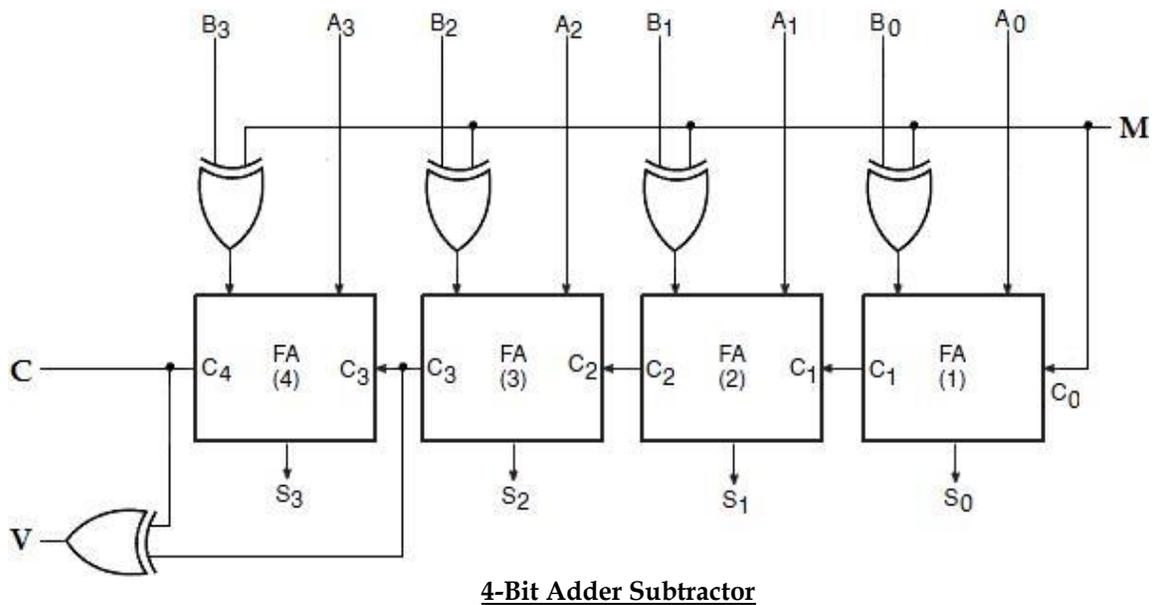
The subtraction of unsigned binary numbers can be done most conveniently by means of complements. The subtraction $A-B$ can be done by taking the 2's complement of B and adding it to A . The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits.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

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 performing subtraction. 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 .



Parallel Adder/ Subtractor:

The addition and subtraction operation can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full adder. A 4-bit adder Subtractor circuit is shown below.



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 $B_0 = B$. 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 $B = 1 \oplus B'$ 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.

Decimal Adder (BCD Adder):

The digital system handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD adder is a circuit that adds two BCD bits and produces a sum digit also in BCD.

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$; the 1 is the sum being an input carry. The adder will form the sum in binary and produce a result that ranges from 0 through 19.

These binary numbers are labeled by symbols K, Z_8, Z_4, Z_2, Z_1 , K is the carry. The columns under the binary sum list the binary values that appear in the outputs of the 4-bit binary adder. The output sum of the two decimal digits must be represented in BCD.

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9

0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 9 (1001), we obtain a non-valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit to detect sum greater than 9 can be determined by simplifying the boolean expression of the given truth table.

Inputs				Output
S ₃	S ₂	S ₁	S ₀	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

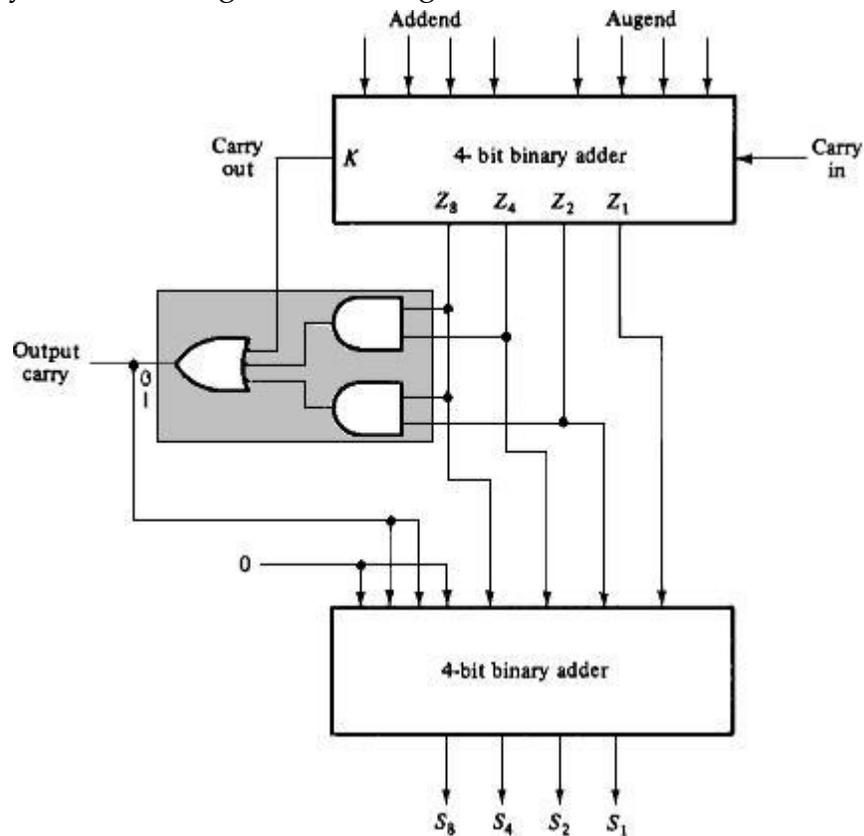
		S ₁ S ₀			
S ₃ S ₂		00	01	11	10
00		0	0	0	0
01		0	0	0	0
11		1	1	1	1
10		0	0	1	1

$$Y = S_3S_2 + S_3S_1$$

To implement BCD adder we require:

- x 4-bit binary adder for initial addition
- x Logic circuit to detect sum greater than 9 and
- x One more 4-bit adder to add 0110₂ in the sum if the sum is greater than 9 or carry is 1.

The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to provide the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. The output carry from one stage must be connected to the input carry of the next higher-order stage.



Block diagram of BCD adder

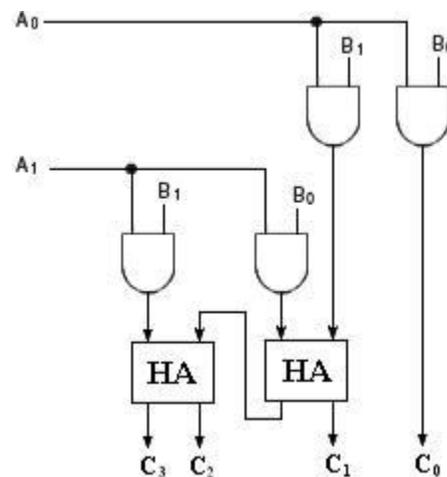
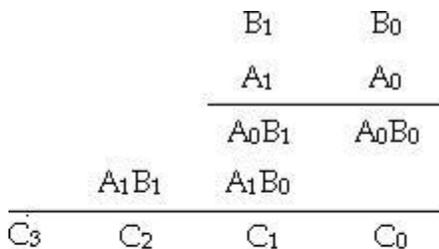
Binary Multiplier:

Multiplication of binary numbers is performed in the same way as in decimal numbers. The multiplicand is multiplied by each bit of the multiplier starting from the least significant bit. Each such multiplication forms a partial product. Such partial

products are shifted one position to the left. The final product is obtained from the sum of partial products.

Consider the multiplication of two 2-bit numbers. The multiplicand bits are B_1 and B_0 , the multiplier bits are A_1 and A_0 , and the product is C_3 , C_2 , C_1 and C_0 . The first partial product is formed by multiplying A_0 by B_1B_0 . The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation. Therefore the partial product can be implemented with AND gates as shown in the diagram below.

The second partial product is formed by multiplying A_1 by B_1B_0 and shifted one position to the left. The two partial products are added with two half adder (HA) circuits.

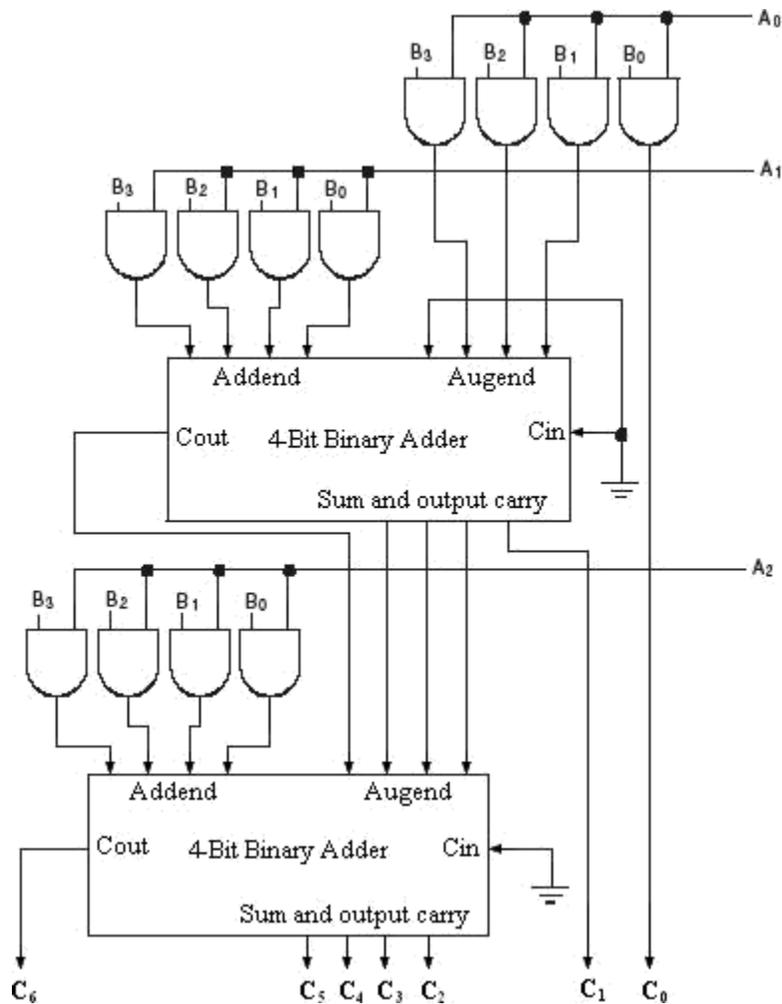


2-bit by 2-bit Binary multiplier

Usually there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products. The least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates are added with the partial product of the previous level to form a new partial product. The last level produces the product. For J multiplier bits and K multiplicand bits we need $(J \times K)$ AND gates and $(J-1)$ k -bit adders to produce a product of $J+K$ bits.

Consider a multiplier circuit that multiplies a binary number of four bits by a number of three bits. Let the multiplicand be represented by B_3 , B_2 , B_1 , B_0 and the multiplier by A_2 , A_1 , and A_0 . Since $K=4$ and $J=3$, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown below.



4-bit by 3-bit Binary multiplier

PARITY GENERATOR/ CHECKER:

A *Parity* is a very useful tool in information processing in digital computers to indicate any presence of error in bit information. External noise and loss of signal strength causes loss of data bit information while transporting data from one device to other device, located inside the computer or externally. To indicate any occurrence of error, an extra bit is included with the message according to the total number of 1s in a set of data, which is called *parity*.

If the extra bit is considered 0 if the total number of 1s is even and 1 for odd quantities of 1s in a set of data, then it is called *even parity*. On the other hand, if the extra bit is 1 for even quantities of 1s and 0 for an odd number of 1s, then it is called *odd parity*.

The message including the parity is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a parity generator and the circuit that checks the parity in the receiver is called a parity checker.

Parity Generator:

A parity generator is a combination logic system to generate the parity bit at the transmitting side. A table illustrates even parity as well as odd parity for a message consisting of three bits.

3-bit Message			Odd Parity bit	Even Parity bit
A	B	C		
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Parity generator truth table for even and odd parity

If the message bit combination is designated as A, B, C and P_e, P_o are the even and odd parity respectively, then it is obvious from table that the boolean expressions of even parity and odd parity are

$$P_e = A \oplus B \oplus C$$

$$P_o = (A \oplus B \oplus C)'$$

K-map Simplification:

		BC			
		00	01	11	10
A	0	0	1	0	1
	1	1	0	1	0

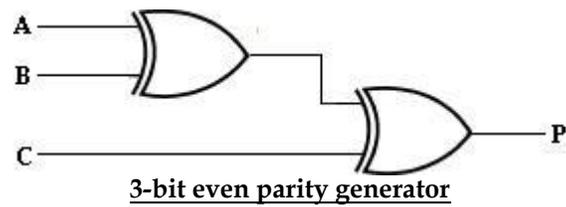
$$P = A'B'C + A'BC' + A'B'C' + ABC$$

$$= A'(B'C + BC') + A(B'C' + BC)$$

$$= A'(B \oplus C) + A(B \oplus C)'$$

$$= A \oplus B \oplus C$$

Logic Diagram:



Parity Checker:

The message bits with the parity bit are transmitted to their destination, where they are applied to a parity checker circuit. The circuit that checks the parity at the receiver side is called the *parity checker*. The parity checker circuit produces a check bit and is very similar to the parity generator circuit. If the check bit is 1, then it is assumed that the received data is incorrect. The check bit will be 0 if the received data is correct. The table shows the truth table for the even parity checker.

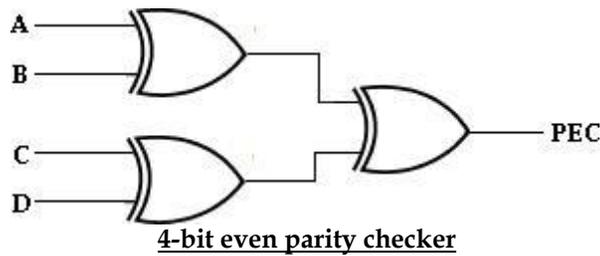
4-Bit Received				Parity Error Check (PEC)
A	B	C	D	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

K-map Simplification:

		CD			
		00	01	11	10
AB	00	0	1	0	1
	01	1	0	1	0
	11	0	1	0	1
	10	1	0	1	0

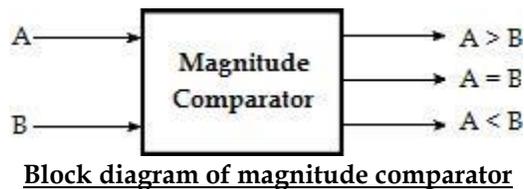
$$\begin{aligned}
 PEC &= A'B'(C'D + CD') + A'B(C'D' + CD) + AB(C'D + CD') + AB'(C'D' + CD) \\
 &= A'B'(C \dagger D) + A'B(C \dagger D)' + AB(C \dagger D) + AB'(C \dagger D)' \\
 &= (A'B' + AB)(C \dagger D) + (A'B + AB')(C \dagger D)' \\
 &= (A \dagger B)'(C \dagger D) + (A \dagger B)(C \dagger D)' \\
 &= (A \dagger B) \dagger (C \dagger D)
 \end{aligned}$$

Logic Diagram:



MAGNITUDE COMPARATOR:

A *magnitude comparator* is a combinational circuit that compares two given numbers (A and B) and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions $A = B$, $A > B$ and $A < B$, if A and B are the two numbers being compared.



For comparison of two n -bit numbers, the classical method to achieve the Boolean expressions requires a truth table of 2^{2n} entries and becomes too lengthy and cumbersome.

2-bit Magnitude Comparator:

The truth table of 2-bit comparator is given in table below –

Truth table:

Inputs				Outputs		
A ₃	A ₂	A ₁	A ₀	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

K-map Simplification:

For A>B

	$B_1 B_0$	00	01	11	10
$A_1 A_0$	00	0	0	0	0
01	00	1	0	0	0
11	00	1	1	0	1
10	00	1	1	0	0

$$A > B = A_0 B_1' B_0' + A_1 B_1' + A_1 A_0 B_0'$$

For A=B

	$B_1 B_0$	00	01	11	10
$A_1 A_0$	00	1	0	0	0
01	00	0	1	0	0
11	00	0	0	1	0
10	00	0	0	0	1

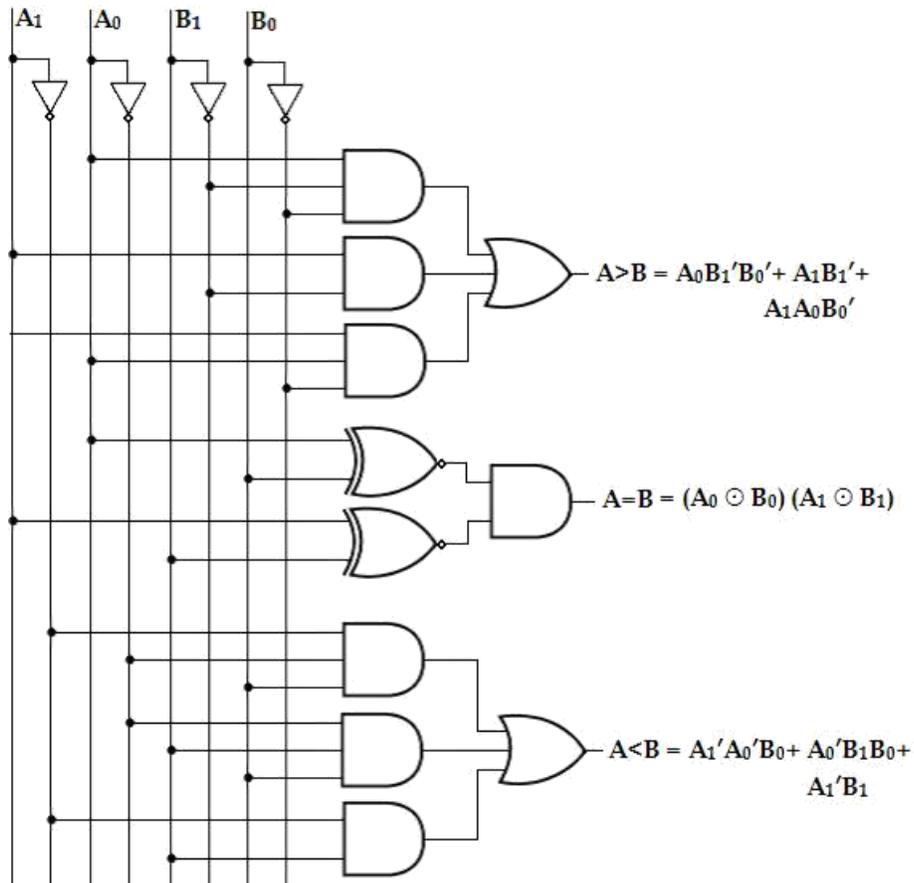
$$\begin{aligned}
 A = B &= A_1' A_0' B_1' B_0' + A_1' A_0 B_1' B_0 + \\
 &\quad A_1 A_0 B_1 B_0 + A_1 A_0' B_1 B_0' \\
 &= A_1' B_1' (A_0' B_0' + A_0 B_0) + A_1 B_1 (A_0 B_0 + A_0' B_0') \\
 &= (A_0 \odot B_0) (A_1 \odot B_1)
 \end{aligned}$$

For A<B

	$B_1 B_0$	00	01	11	10
$A_1 A_0$	00	0	1	1	1
01	00	0	0	1	1
11	00	0	0	0	0
10	00	0	0	1	0

$$A < B = A_1' A_0' B_0 + A_0' B_1 B_0 + A_1' B_1$$

Logic Diagram:



2-bit Magnitude Comparator

4-bit Magnitude Comparator:

Let us consider the two binary numbers A and B with four digits each. Write the coefficient of the numbers in descending order as,

$$A = A_3A_2A_1A_0$$

$$B = B_3 B_2 B_1 B_0,$$

Each subscripted letter represents one of the digits in the number. It is observed from the bit contents of two numbers that $A = B$ when $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$. When the numbers are binary they possess the value of either 1 or 0, the equality relation of each pair can be expressed logically by the equivalence function as

$$X_i = A_i B_i + A_i' B_i' \quad \text{for } i = 1, 2, 3, 4.$$

Or, $X_i = (A \dagger B)'$ or, $X_i' = A \dagger B$

Or, $X_i = (A_i B_i' + A_i' B_i)'$.

where,

$X_i = 1$ only if the pair of bits in position i are equal (ie., if both are 1 or both are 0).

To satisfy the equality condition of two numbers A and B , it is necessary that all X_i must be equal to logic 1. This indicates the AND operation of all X_i variables. In other words, we can write the Boolean expression for two equal 4-bit numbers.

$$(A = B) = X_3 X_2 X_1 X_0.$$

The binary variable $(A=B)$ is equal to 1 only if all pairs of digits of the two numbers are equal.

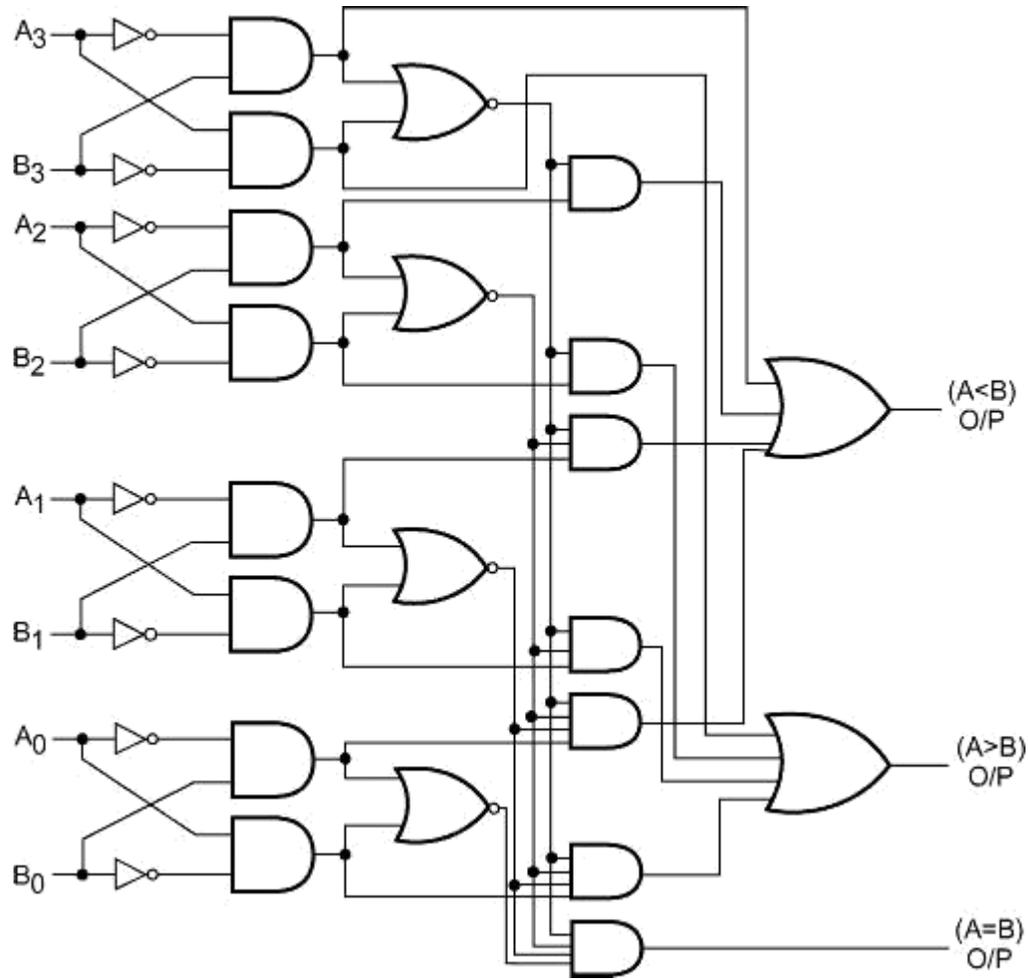
To determine if A is greater than or less than B , we inspect the relative magnitudes of pairs of significant bits starting from the most significant bit. If the two digits of the most significant position are equal, the next significant pair of digits is compared. The comparison process is continued until a pair of unequal digits is found. It may be concluded that $A > B$, if the corresponding digit of A is 1 and B is 0. If the corresponding digit of A is 0 and B is 1, we conclude that $A < B$. Therefore, we can derive the logical expression of such sequential comparison by the following two Boolean functions,

$$(A > B) = A_3 B_3' + X_3 A_2 B_2' + X_3 X_2 A_1 B_1' + X_3 X_2 X_1 A_0 B_0'$$

$$(A < B) = A_3' B_3 + X_3 A_2' B_2 + X_3 X_2 A_1' B_1 + X_3 X_2 X_1 A_0' B_0$$

The symbols $(A > B)$ and $(A < B)$ are binary output variables that are equal to 1 when $A > B$ or $A < B$, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The unequal outputs can use the same gates that are needed to generate the equal output. The logic diagram of the 4-bit magnitude comparator is shown below,



4-bit Magnitude Comparator

The four x outputs are generated with exclusive-NOR circuits and applied to an AND gate to give the binary output variable $(A=B)$. The other two outputs use the x variables to generate the Boolean functions listed above. This is a multilevel implementation and has a regular pattern.

CODE CONVERTERS:

A code converter is a logic circuit that changes data presented in one type of binary code to another code of binary code. The following are some of the most commonly used code converters:

- i. Binary-to-Gray code
- ii. Gray-to-Binary code
- iii. BCD-to-Excess-3
- iv. Excess-3-to-BCD
- v. Binary-to-BCD
- vi. BCD-to-binary
- vii. Gray-to-BCD
- viii. BCD-to-Gray
- ix. 8 4 -2 -1 to BCD converter

1. Binary to Gray Converters:

The gray code is often used in digital systems because it has the advantage that only one bit in the numerical representation changes between successive numbers. The truth table for the binary-to-gray code converter is shown below,

Truth table:

Decimal	Binary code				Gray code			
	B ₃	B ₂	B ₁	B ₀	G ₃	G ₂	G ₁	G ₀
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

K-map simplification:

For G₃

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	0	0	0
	01	0	0	0	0
	11	1	1	1	1
	10	1	1	1	1

$$G_3 = B_3$$

For G₂

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	0	0	0
	01	1	1	1	1
	11	0	0	0	0
	10	1	1	1	1

$$G_2 = B_3' B_2 + B_3 B_2'$$

$$= B_3 \oplus B_2$$

For G₁

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	0	1	1
	01	1	1	0	0
	11	1	1	0	0
	10	0	0	1	1

$$G_1 = B_2' B_1 + B_2 B_1'$$

$$= B_2 \oplus B_1$$

For G₀

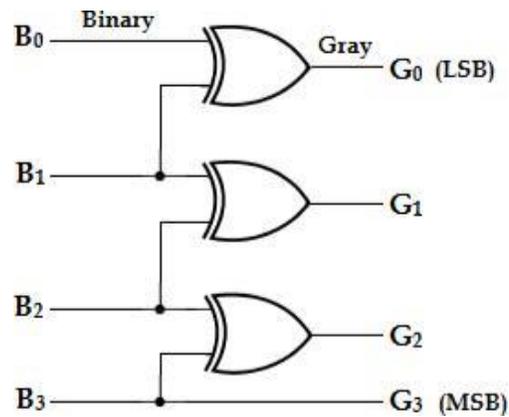
		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	1	0	1
	01	0	1	0	1
	11	0	1	0	1
	10	0	1	0	1

$$G_0 = B_1' B_0 + B_1 B_0'$$

$$= B_1 \oplus B_0$$

Now, the above expressions can be implemented using EX-OR gates as,

Logic Diagram:



2. Gray to Binary Converters:

The truth table for the gray-to-binary code converter is shown below,

Truth table:

Gray code				Binary code			
G ₃	G ₂	G ₁	G ₀	B ₃	B ₂	B ₁	B ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

From the truth table, the logic expression for the binary code outputs can be written as,

$$G_3 = \sum_m (8, 9, 10, 11, 12, 13, 14, 15)$$

$$G_2 = \sum_m (4, 5, 6, 7, 8, 9, 10, 11)$$

$$G_1 = \sum_m (2, 3, 4, 5, 8, 9, 14, 15)$$

$$G_0 = \sum_m (1, 2, 4, 7, 8, 11, 13, 14)$$

K-map Simplification:

For B₃

		$G_1 G_0$			
		00	01	11	10
$G_3 G_2$	00	0	0	0	0
	01	0	0	0	0
	11	1	1	1	1
	10	1	1	1	1

$$B_3 = G_3$$

For B₂

		$G_1 G_0$			
		00	01	11	10
$G_3 G_2$	00	0	0	0	0
	01	1	1	1	1
	11	0	0	0	0
	10	1	1	1	1

$$B_2 = G_3'G_2 + G_3G_2'$$

$$= G_3 \oplus G_2$$

For B₁

		$G_1 G_0$			
		00	01	11	10
$G_3 G_2$	00	0	0	1	1
	01	1	1	0	0
	11	0	0	1	1
	10	1	1	0	0

For B₀

		$G_1 G_0$			
		00	01	11	10
$G_3 G_2$	00	0	1	0	1
	01	1	0	1	0
	11	0	1	0	1
	10	1	0	1	0

From the above K-map,

$$B_3 = G_3$$

$$B_2 = G_3'G_2 + G_3G_2'$$

$$B_2 = G_3 \uparrow G_2$$

$$B_1 = G_3'G_2'G_1 + G_3'G_2G_1' + G_3G_2G_1 + G_3G_2'G_1'$$

$$= G_3' (G_2'G_1 + G_2G_1') + G_3 (G_2G_1 + G_2'G_1')$$

$$= G_3' (G_2 \uparrow G_1) + G_3 (G_2 \uparrow G_1)' \quad [x \uparrow y = x'y + xy'], [(x \uparrow y)' = xy + x'y']$$

$$B_1 = G_3 \uparrow G_2 \uparrow G_1$$

$$B_0 = G_3'G_2'G_1'G_0 + G_3'G_2'G_1G_0' + G_3'G_2G_1'G_0' + G_3'G_2G_1G_0 + G_3G_2'G_1'G_0' +$$

$$G_3G_2'G_1G_0' + G_3G_2G_1'G_0 + G_3G_2G_1G_0'$$

$$= G_3'G_2' (G_1'G_0 + G_1G_0') + G_3'G_2 (G_1'G_0' + G_1G_0) + G_1'G_0' (G_3'G_2 + G_3G_2') + G_1G_0 (G_3'G_2 + G_3G_2')$$

$$= G_3'G_2' (G_0 \uparrow G_1) + G_3'G_2 (G_0 \uparrow G_1) + G_1'G_0' (G_2 \uparrow G_3) + G_1G_0 (G_2 \uparrow G_3)$$

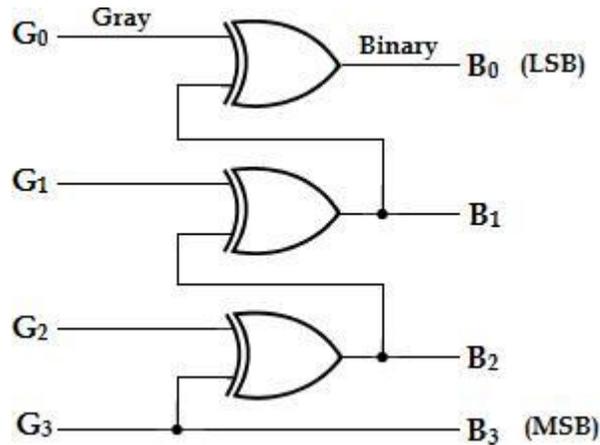
$$= G_0 \uparrow G_1 (G_3'G_2' + G_3G_2) + G_2 \uparrow G_3 (G_1'G_0' + G_1G_0)$$

$$= (G_0 \uparrow G_1) (G_2 \uparrow G_3)' + (G_2 \uparrow G_3) (G_0 \uparrow G_1)$$

$$[x \uparrow y = x'y + xy']$$

$$B_0 = (G_0 \uparrow G_1) \uparrow (G_2 \uparrow G_3)$$

Now, the above expressions can be implemented using EX-OR gates as,



Logic diagram of 4-bit gray-to-binary converter

3. BCD -to-Excess-3 Converters:

Excess-3 is a modified form of a BCD number. The excess-3 code can be derived from the natural BCD code by adding 3 to each coded number.

For example, decimal 12 can be represented in BCD as 0001 0010. Now adding 3 to each digit we get excess-3 code as 0100 0101 (12 in decimal). With this information the truth table for BCD to Excess-3 code converter can be determined as,

Truth Table:

Decimal	BCD code				Excess-3 code			
	B ₃	B ₂	B ₁	B ₀	E ₃	E ₂	E ₁	E ₀
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$$E_3 = \sum_m (5, 6, 7, 8, 9) + \sum_d (10, 11, 12, 13, 14, 15)$$

$$E_2 = \sum_m (1, 2, 3, 4, 9) + \sum_d (10, 11, 12, 13, 14, 15)$$

$$E_1 = \sum_m (0, 3, 4, 7, 8) + \sum_d (10, 11, 12, 13, 14, 15)$$

$$E_0 = \sum_m (0, 2, 4, 6, 8) + \sum_d (10, 11, 12, 13, 14, 15)$$

K-map Simplification:

For E₃

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	0	0	0
	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

$$E_3 = B_3 + B_2 (B_0 + B_1)$$

For E₂

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	0	1	1	1
	01	1	0	0	0
	11	x	x	x	x
	10	0	1	x	x

$$E_2 = B_2 B_1' B_0' + B_2' (B_0 + B_1)$$

For E₁

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	1	0	1	0
	01	1	0	1	0
	11	x	x	x	x
	10	1	0	x	x

$$E_1 = B_1' B_0' + B_1 B_0$$

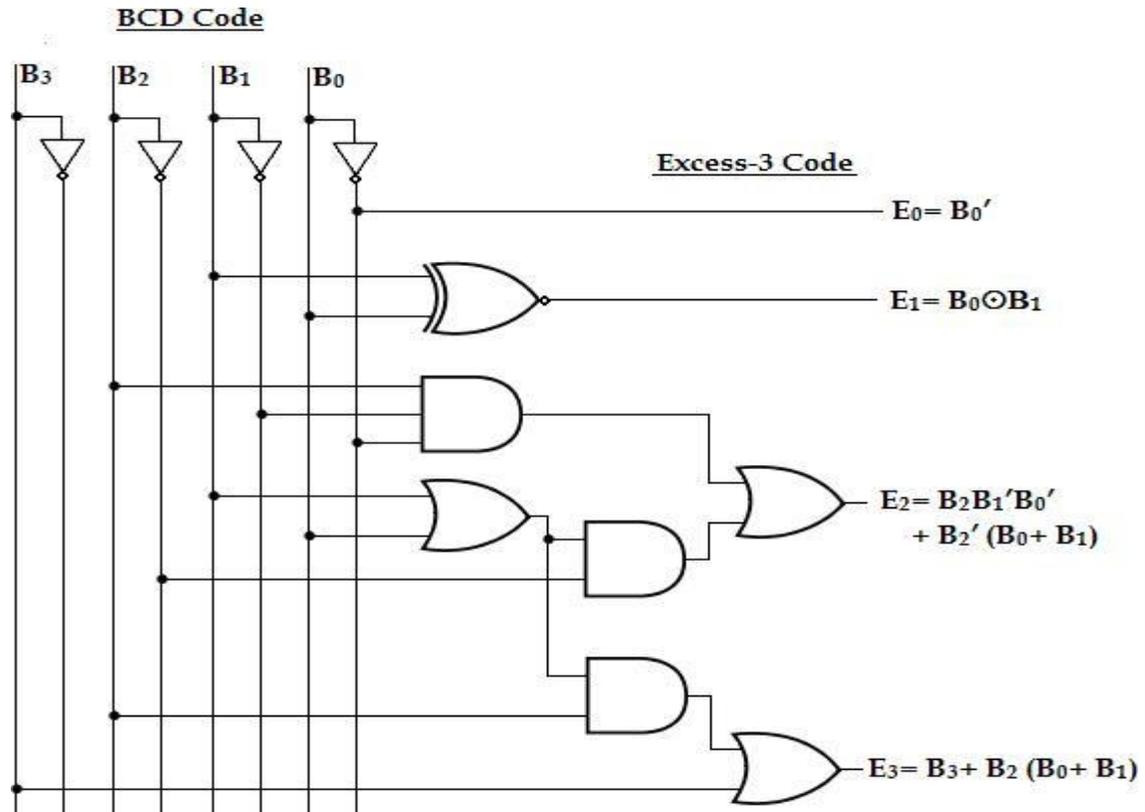
$$= B_1 \odot B_0$$

For E₀

		$B_1 B_0$			
		00	01	11	10
$B_3 B_2$	00	1	0	0	1
	01	1	0	0	1
	11	x	x	x	x
	10	1	0	x	x

$$E_0 = B_0'$$

Logic Diagram:



4. Excess-3 to BCD

Converter: Truth table:

Decimal	Excess-3 code				BCD code			
	E ₃	E ₂	E ₁	E ₀	B ₃	B ₂	B ₁	B ₀
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	1
11	1	0	1	1	1	0	0	0
12	1	1	0	0	1	0	0	1

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$$B_3 = \sum_m (11, 12) + \sum_d (0, 1, 2, 13, 14, 15)$$

$$B_2 = \sum_m (7, 8, 9, 10) + \sum_d (0, 1, 2, 13, 14, 15)$$

$$B_1 = \sum_m (5, 6, 9, 10) + \sum_d (0, 1, 2, 13, 14, 15)$$

$$B_0 = \sum_m (4, 6, 8, 10, 12) + \sum_d (0, 1, 2, 13, 14, 15)$$

K-map Simplification:

For B₃

		$E_1 E_0$			
		00	01	11	10
$E_3 E_2$	00	X	X	0	X
	01	0	0	0	0
	11	1	X	X	X
	10	0	0	1	0

$$B_3 = E_3 E_2 + E_3 E_1 E_0$$

For B₂

		$E_1 E_0$			
		00	01	11	10
$E_3 E_2$	00	X	X	0	X
	01	0	0	1	0
	11	0	X	X	X
	10	1	1	0	1

$$B_2 = E_2' E_1' + E_2 E_1 E_0 + E_3 E_1 E_0'$$

For B₁

		$E_1 E_0$			
		00	01	11	10
$E_3 E_2$	00	X	X	0	X
	01	0	1	0	1
	11	0	X	X	X
	10	0	1	0	1

$$B_1 = E_1' E_0 + E_1 E_0'$$

$$= E_1 \oplus E_0$$

For B₀

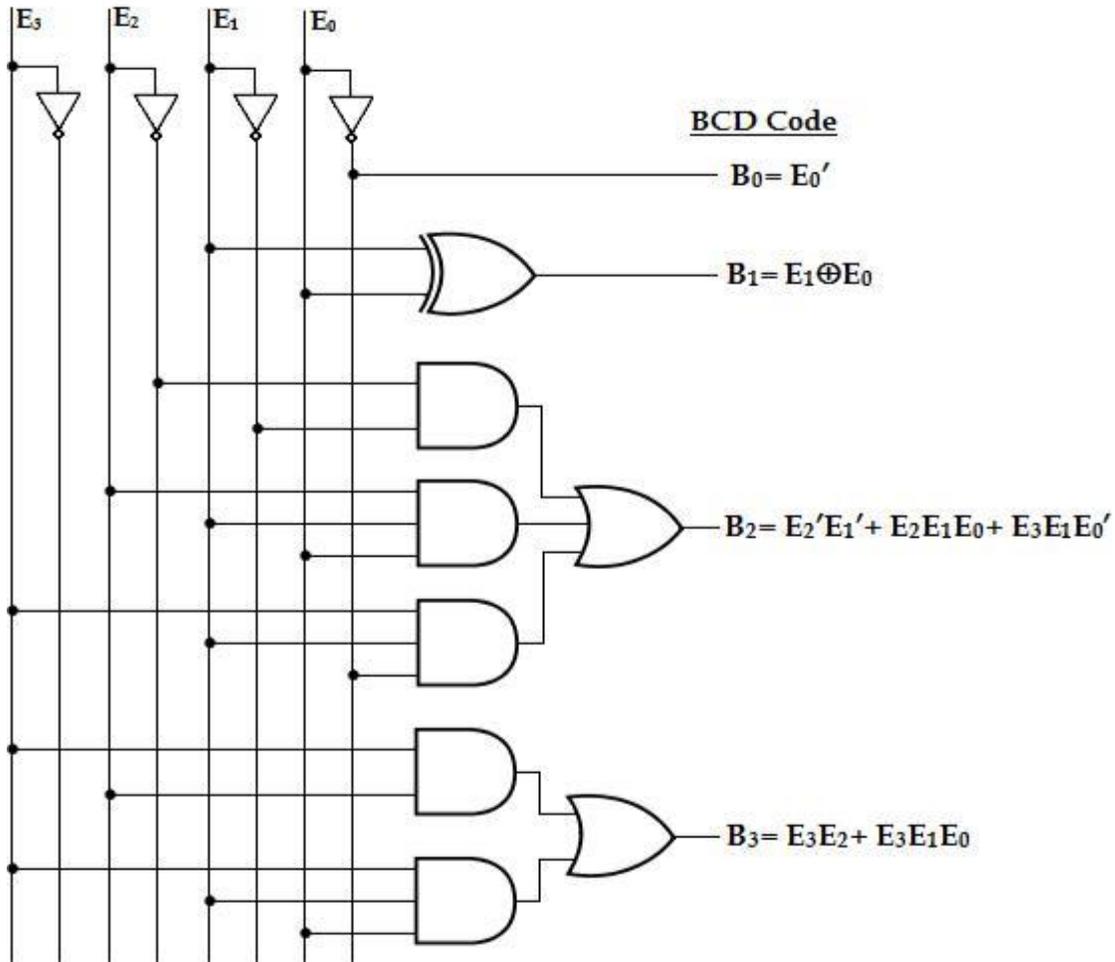
		$E_1 E_0$			
		00	01	11	10
$E_3 E_2$	00	X	X	0	X
	01	1	0	0	1
	11	1	X	X	X
	10	1	0	0	1

$$B_0 = E_0'$$

Now, the above expressions the logic diagram can be implemented as,

Logic Diagram:

Excess-3 Code

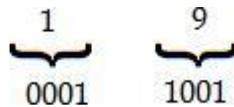


5. BCD -to-Binary Converters:

The steps involved in the BCD-to-binary conversion process are as follows:

1. The value of each bit in the BCD number is represented by a binary equivalent or weight.
2. All the binary weights of the bits that are 1's in the BCD are added.
3. The result of this addition is the binary equivalent of the BCD number.

Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two 4-bit code groups. For example, 19_{10} is represented as,

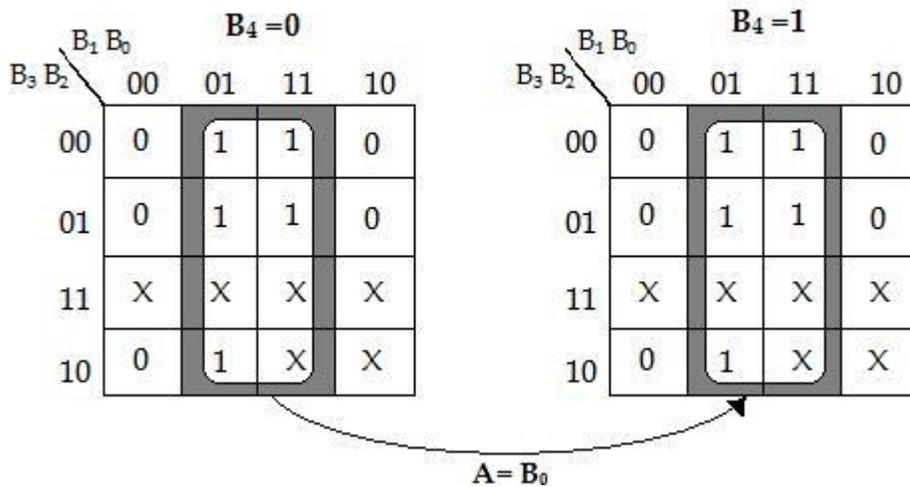


The left-most four-bit group represents 10 and right-most four-bit group represents 9. The binary representation for decimal 19 is $19_{10} = 11001_2$.

BCD Code					Binary				
B ₄	B ₃	B ₂	B ₁	B ₀	E	D	C	B	A
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
1	0	0	0	0	0	1	0	1	0
1	0	0	0	1	0	1	0	1	1
1	0	0	1	0	0	1	1	0	0
1	0	0	1	1	0	1	1	0	1
1	0	1	0	0	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	0	0	1	0
1	1	0	0	1	1	0	0	1	1

K-map Simplification:

For A



For B

$B_3 B_2$		$B_4 = 0$			
		$B_1 B_0$	00	01	11
00	00	0	0	1	1
01	01	0	0	1	1
11	11	X	X	X	X
10	10	0	0	X	X

$B_3 B_2$		$B_4 = 1$			
		$B_1 B_0$	00	01	11
00	00	1	1	0	0
01	01	1	1	0	0
11	11	X	X	X	X
10	10	1	1	X	X

$$B = B_1 B_4' + B_1' B_4$$

$$= B_1 \oplus B_4$$

For C

$B_3 B_2$		$B_4 = 0$			
		$B_1 B_0$	00	01	11
00	00	0	0	0	0
01	01	1	1	1	1
11	11	X	X	X	X
10	10	0	0	X	X

$B_3 B_2$		$B_4 = 1$			
		$B_1 B_0$	00	01	11
00	00	0	0	1	1
01	01	1	1	0	0
11	11	X	X	X	X
10	10	0	0	X	X

$$C = B_4' B_2 + B_2 B_1' + B_4 B_2' B_1$$

For D

$B_3 B_2$		$B_4 = 0$			
		$B_1 B_0$	00	01	11
00	00	0	0	0	0
01	01	0	0	0	0
11	11	X	X	X	X
10	10	1	1	X	X

$B_3 B_2$		$B_4 = 1$			
		$B_1 B_0$	00	01	11
00	00	1	1	1	1
01	01	1	1	0	0
11	11	X	X	X	X
10	10	0	0	X	X

$$D = B_4' B_3 + B_4 B_3' B_2' + B_4 B_3' B_1'$$

For E

$B_3 B_2$		$B_4 = 0$				$B_3 B_2$		$B_4 = 1$			
		$B_1 B_0$	00	01	11			10	$B_1 B_0$	00	01
00	00	0	0	0	0	00	0	0	0	0	
01	00	0	0	0	0	01	0	0	1	1	
11	00	X	X	X	X	11	X	X	X	X	
10	00	0	0	X	X	10	1	1	X	X	

$$E = B_4 B_3 + B_4 B_2 B_1$$

From the above K-map,

$$A = B_0$$

$$\begin{aligned} B &= B_1 B_4' + B_1' B_4 \\ &= B_1 \uparrow B_4 \end{aligned}$$

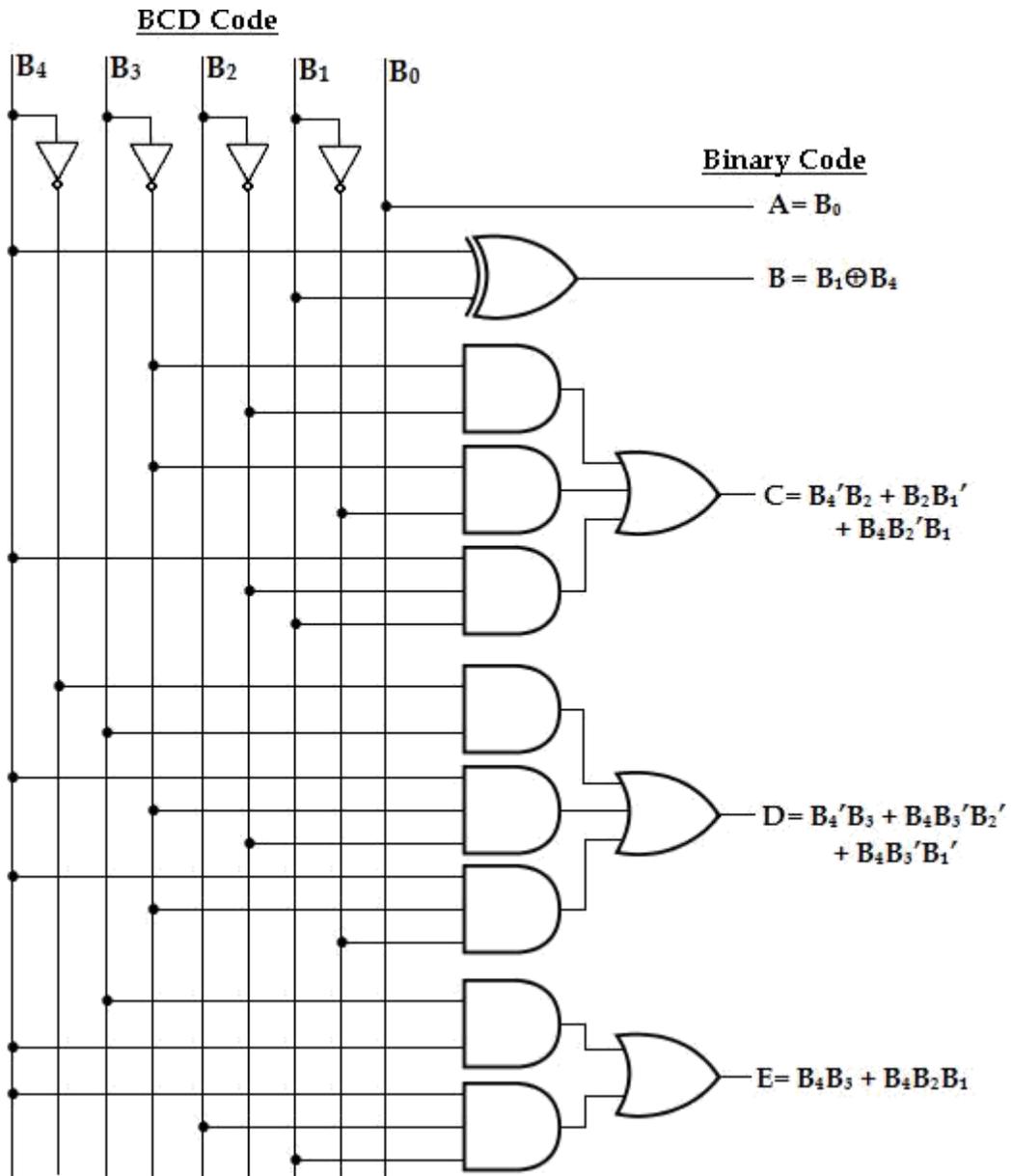
$$C = B_4' B_2 + B_2 B_1' + B_4 B_2' B_1$$

$$D = B_4' B_3 + B_4 B_3' B_2' + B_4 B_3' B_1'$$

$$E = B_4 B_3 + B_4 B_2 B_1$$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



6. Binary to BCD Converter:

The truth table for binary to BCD converter can be written as,

Truth Table:

Decimal	Binary Code				BCD Code				
	D	C	B	A	B ₄	B ₃	B ₂	B ₁	B ₀
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	0	1

From the truth table, the logic expression for the BCD code outputs can be written as,

$$B_0 = \sum_m (1, 3, 5, 7, 9, 11, 13, 15)$$

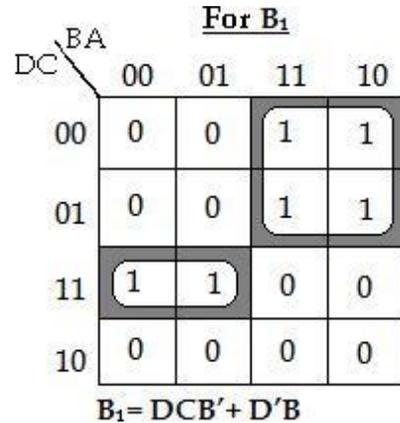
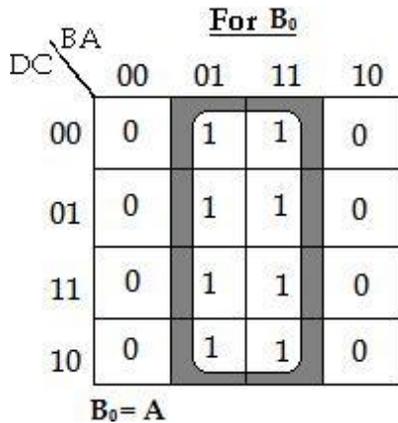
$$B_1 = \sum_m (2, 3, 6, 7, 12, 13)$$

$$B_2 = \sum_m (4, 5, 6, 7, 14, 15)$$

$$B_3 = \sum_m (8, 9)$$

$$B_4 = \sum_m (10, 11, 12, 13, 14, 15)$$

K-map Simplification:



For B_2

DC \ BA	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	1	1
10	0	0	0	0

$$B_2 = D'C + CB$$

For B_3

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	1	0	0

$$B_3 = DC'B'$$

For B_4

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$$B_4 = DC + DB$$

From the above K-map, the logical expression can be obtained as, $B_0 = A$

$$B_1 = DCB' + D'B$$

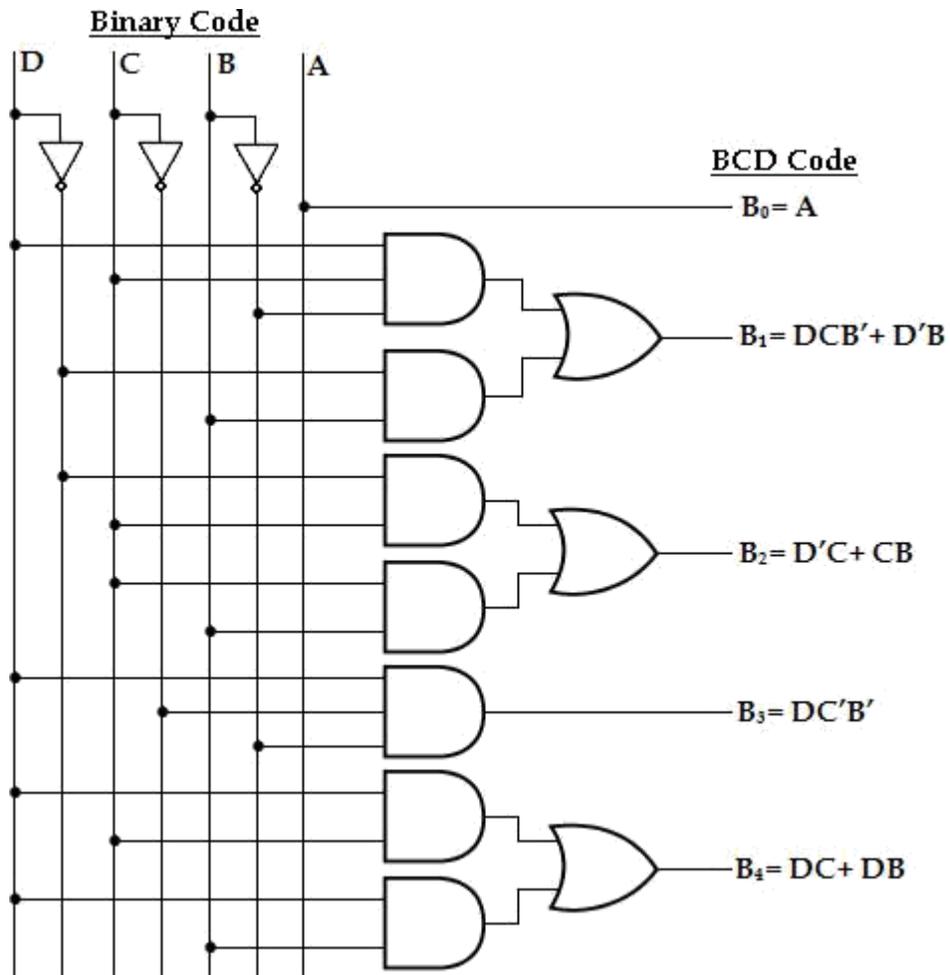
$$B_2 = D'C + CB$$

$$B_3 = DC'B'$$

$$B_4 = DC + DB$$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



7. Gray to BCD Converter:

The truth table for gray to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
G ₃	G ₂	G ₁	G ₀	B ₄	B ₃	B ₂	B ₁	B ₀
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	1	1	0	0	0	1	0	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	0	0	0	0	1	1	1
1	1	0	0	0	1	0	0	0

1	1	0	1	0	1	0	0	1
1	1	1	1	1	0	0	0	0
1	1	1	0	1	0	0	0	1
1	0	1	0	1	0	0	1	0
1	0	1	1	1	0	0	1	1
1	0	0	1	1	0	1	0	0
1	0	0	0	1	0	1	0	1

K-map Simplification:

For B₀

$G_3 G_2 \backslash G_1 G_0$	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$B_0 = (G_0 \oplus G_1) \oplus (G_2 \oplus G_3)$

For B₁

$G_3 G_2 \backslash G_1 G_0$	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	1	1

$B_1 = G'_2 G_1 + G'_3 G_2 G'_1$

For B₂

$G_3 G_2 \backslash G_1 G_0$	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	0	0

$B_2 = G'_3 G_2 + G_3 G'_2 G'_1$

For B₃

$G_3 G_2 \backslash G_1 G_0$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	0	0
10	0	0	0	0

$B_3 = G_3 G_2 G'_1$

		<u>For B₄</u>			
		G ₁ G ₀	00	01	11
G ₃ G ₂	00	0	0	0	0
	01	0	0	0	0
	11	0	0	1	1
	10	1	1	1	1

$$B_4 = G_3 G'_2 + G_3 G_1$$

From the above K-map, the logical expression can be obtained as,

$$B_0 = (G_0 + G_1) + (G_2 + G_3)$$

$$B_1 = G'_2 G_1 + G'_3 G_2 G'_1$$

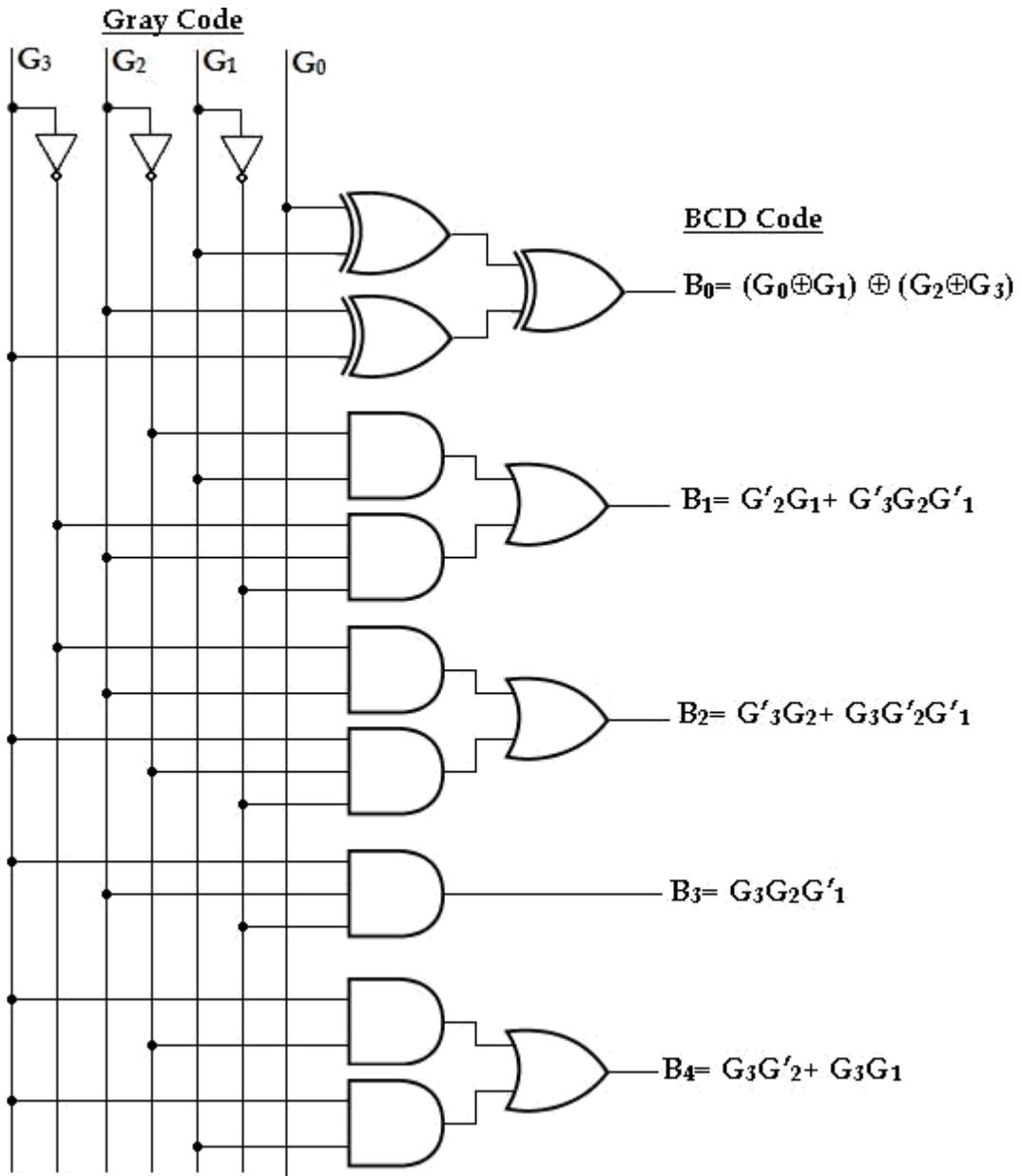
$$B_2 = G'_3 G_2 + G_3 G'_2 G'_1$$

$$B_3 = G_3 G_2 G'_1$$

$$B_4 = G_3 G'_2 + G_3 G_1$$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



8. BCD to Gray Converter:

The truth table for gray to BCD converter can be written as,

Truth table:

BCD Code (8421)				Gray code			
B ₃	B ₂	B ₁	B ₀	G ₃	G ₂	G ₁	G ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0

0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

K-map Simplification:

For G₃

B ₃ B ₂ \ B ₁ B ₀	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

$G_3 = B_3$

For G₂

B ₃ B ₂ \ B ₁ B ₀	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

$G_2 = B_3 + B_2$

For G₁

B ₃ B ₂ \ B ₁ B ₀	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$G_1 = B_2'B_1 + B_2B_1'$
 $= B_2 \oplus B_1$

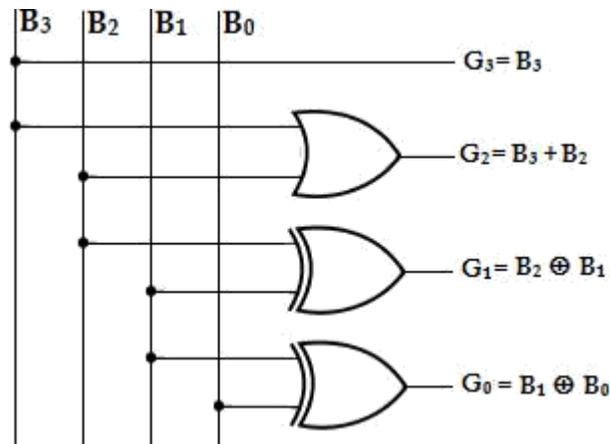
For G₀

B ₃ B ₂ \ B ₁ B ₀	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	X	X	X	X
10	0	1	X	X

$G_0 = B_1'B_0 + B_1B_0'$
 $= B_1 \oplus B_0$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



9. 84 -2 -1 to BCD Converter:

The truth table for 84 -2 -1 to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
D	C	B	A	B ₄	B ₃	B ₂	B ₁	B ₀
0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	1
0	1	1	0	0	0	0	1	0
0	1	0	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	1
1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	1	1	1	0	1	0	0	1
1	1	1	0	1	0	0	0	0
1	1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0

K-map Simplification:

For B₀

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$$B_0 = A$$

For B₁

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	0	1
11	1	0	0	0
10	0	1	0	1

$$\begin{aligned} B_1 &= DCB'A' + D'B'A + D'BA' + C'B'A + C'BA' \\ &= A'B'CD + D'(B'A + BA) + C'(B'A + BA) \\ &= A'B'CD + D'(A \oplus B) + C'(A \oplus B) \\ &= A'B'CD + (A \oplus B)(C' + D') \end{aligned}$$

For B₂

DC \ BA	00	01	11	10
00	0	X	X	X
01	1	0	0	0
11	0	0	0	0
10	0	1	1	1

$$\begin{aligned} B_2 &= D'CB'A' + C'A + C'B \\ &= D'CB'A' + C'(A + B) \end{aligned}$$

For B₃

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	0	0	1	0
10	1	0	0	0

$$\begin{aligned} B_3 &= ABCD + A'B'C'D \\ &= D(ABC + A'B'C) \end{aligned}$$

For B₄

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	1	1	0	1
10	0	0	0	0

$$\begin{aligned} B_4 &= B'CD + A'CD \\ &= CD(A' + B') \end{aligned}$$

From the above K-map, the logical expression can be obtained as,

$$B_0 = A$$

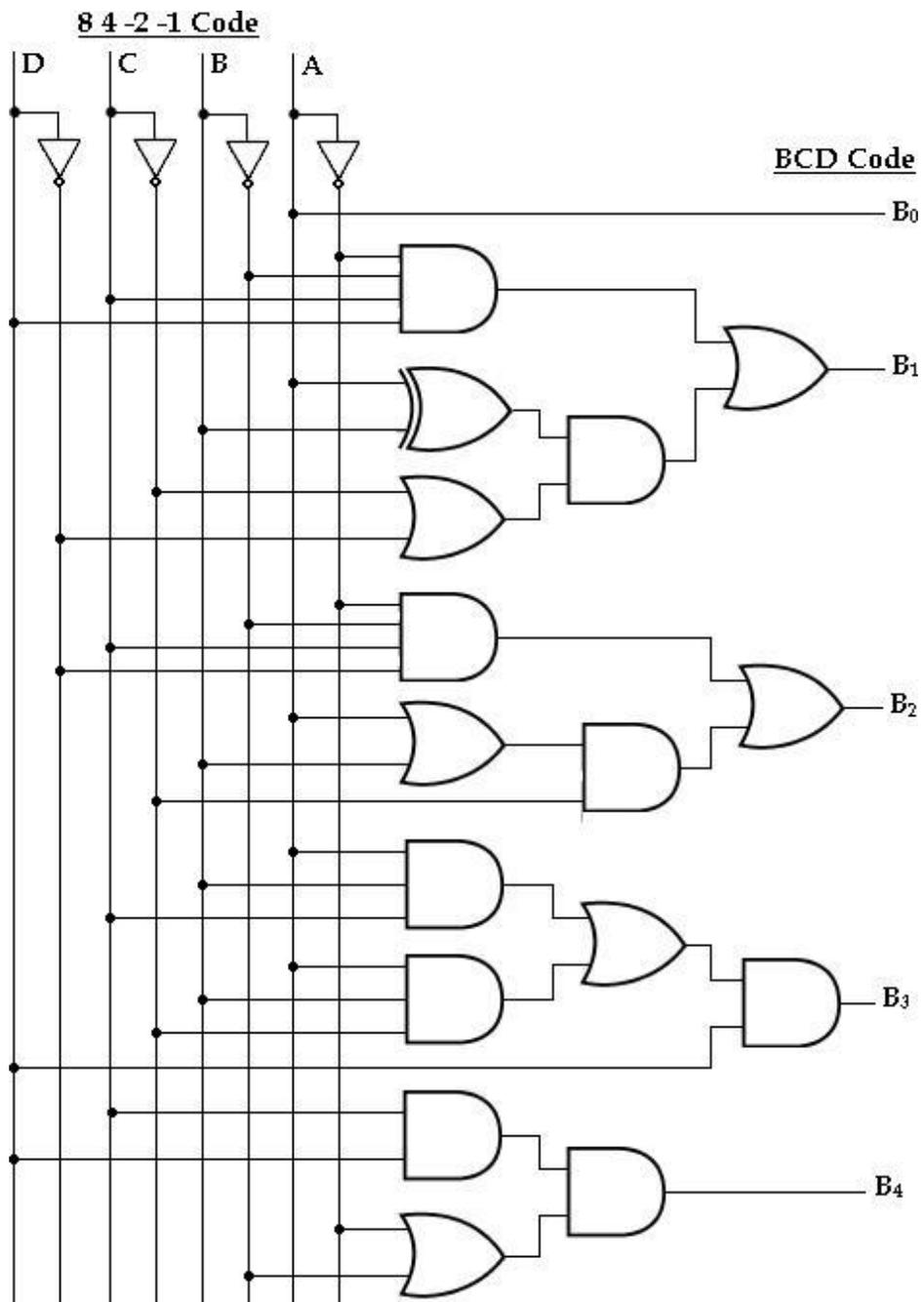
$$B_1 = A'B'CD + (A+B)(C'+D')$$

$$B_2 = D'CB'A' + C'(A+B)$$

$$B_3 = D(ABC + A'B'C')$$

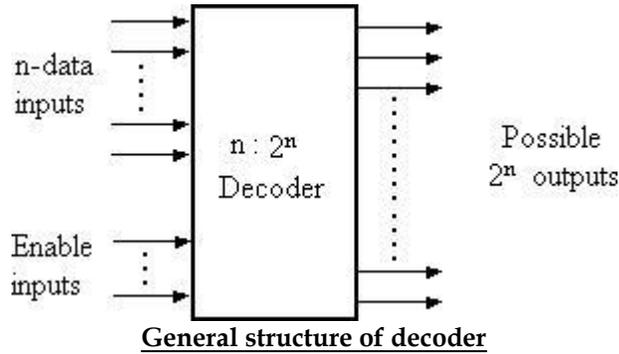
$$B_4 = CD(A'+B')$$

Logic Diagram:



DECODERS:

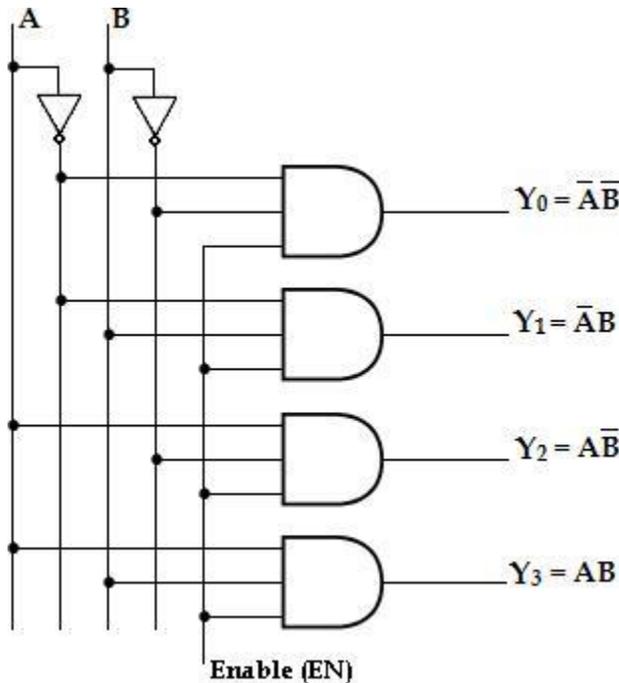
A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. The general structure of decoder circuit is -

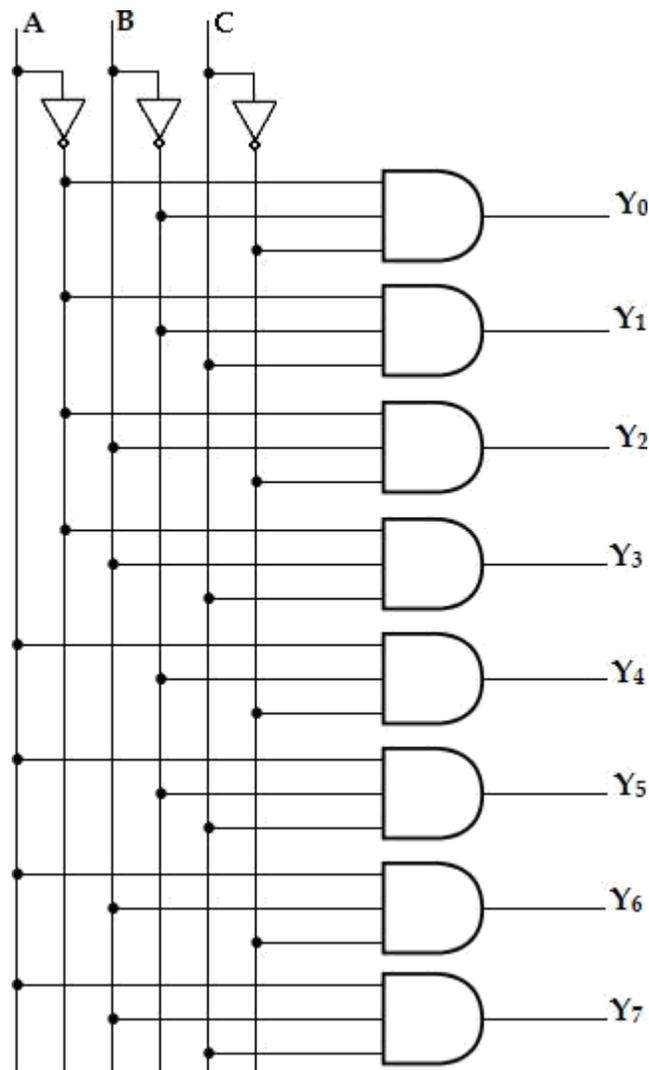


The encoded information is presented as n inputs producing 2^n possible outputs. The 2^n output values are from 0 through $2^n - 1$. A decoder is provided with enable inputs to activate decoded output based on data inputs. When any one enable input is unasserted, all outputs of decoder are disabled.

Binary Decoder (2 to 4 decoder):

A binary decoder has n bit binary input and a one activated output out of 2^n outputs. A binary decoder is used when it is necessary to activate exactly one of 2^n outputs based on an n -bit input value.

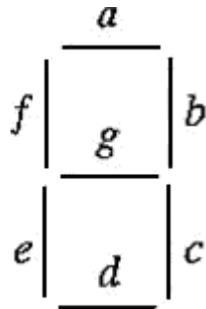




3-to-8 line decoder

BCD to 7-Segment Display Decoder:

A seven-segment display is normally used for displaying any one of the decimal digits, 0 through 9. A BCD-to-seven segment decoder accepts a decimal digit in BCD and generates the corresponding seven-segment code.



Each segment is made up of a material that emits light when current is passed through it. The segments activated during each digit display are tabulated as –

Digit	Display	Segments Activated
0		a, b, c, d, e, f
1		b, c
2		a, b, d, e, g
3		a, b, c, d, g
4		b, c, f, g
5		a, c, d, f, g

6		a, c, d, e, f, g
7		a, b, c
8		a, b, c, d, e, f, g
9		a, b, c, d, f, g

Truth table:

Digit	BCD code				7-Segment code						
	A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

K-map Simplification:

For (a)

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$a = A + C + BD + B'D'$

For (b)

AB \ CD	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	X	X	X	X
10	1	1	X	X

$b = B' + C'D' + CD$

For (c)

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

$c = B + C' + D$

For (d)

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	X	X	X	X
10	1	1	X	X

$d = B'D' + CD' + BC'D + B'C + A$

For (e)

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	X	X	X	X
10	1	0	X	X

$e = B'D' + CD'$

For (f)

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

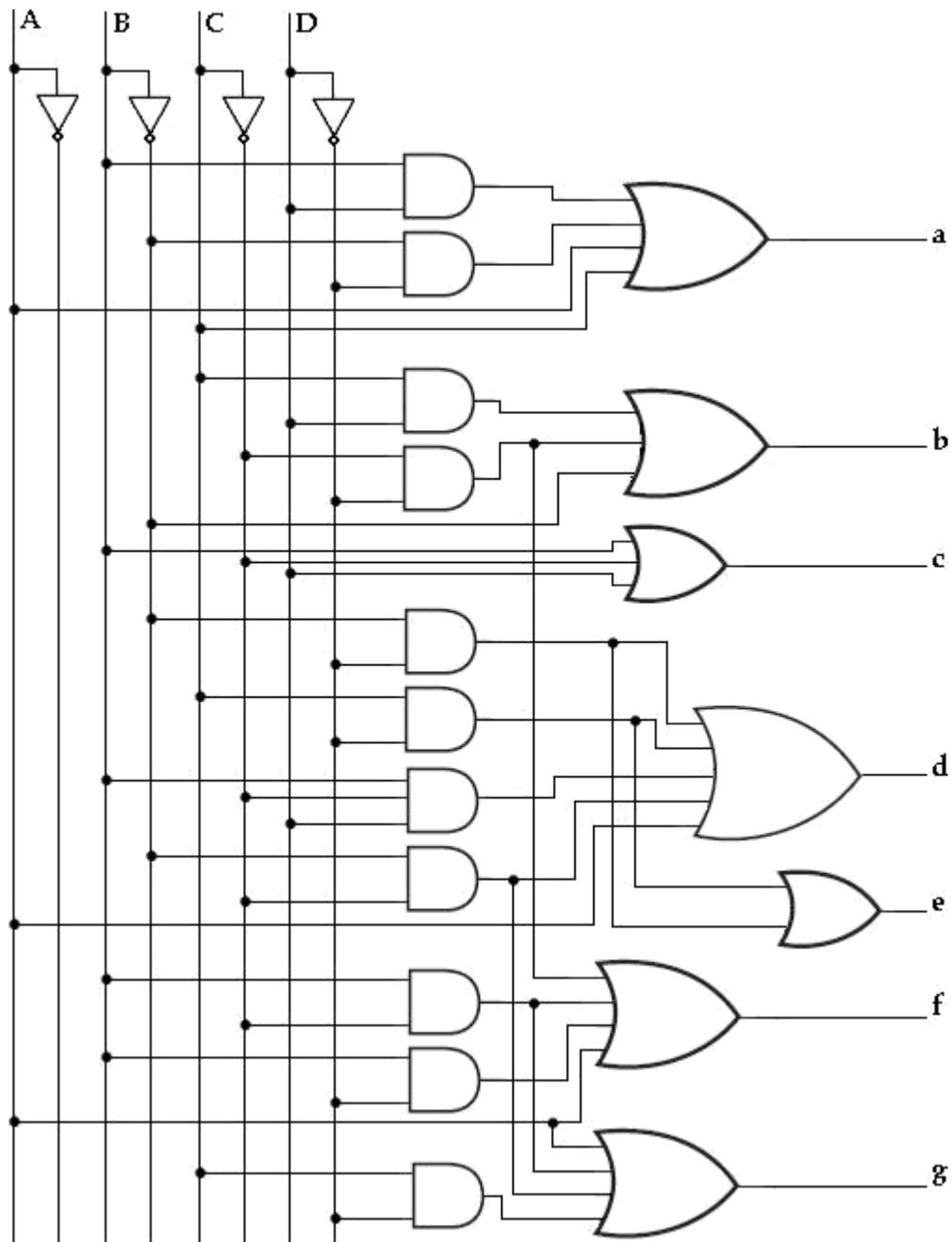
$f = A + C'D' + BC' + BD'$

For (g)

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

$g = A + BC' + B'C + CD'$

Logic Diagram:



BCD to 7-segment display decoder

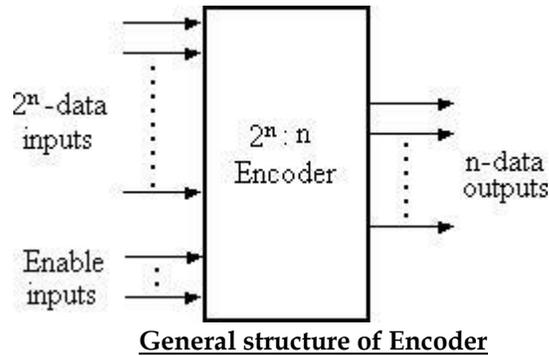
Applications of decoders:

1. Decoders are used in counter system.
2. They are used in analog to digital converter.
3. Decoder outputs can be used to drive a display system.

ENCODERS:

An encoder is a digital circuit that performs the inverse operation of a decoder. Hence, the opposite of the decoding process is called encoding. An encoder is a combinational circuit that converts binary information from 2^n input lines to a maximum of n' unique output lines.

The general structure of encoder circuit is -



It has 2^n input lines, only one which 1 is active at any time and n' output lines. It encodes one of the active inputs to a coded binary output with n' bits. In an encoder, the number of outputs is less than the number of inputs.

Octal-to-Binary Encoder:

It has eight inputs (one for each of the octal digits) and the three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

Inputs								Outputs		
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	A	B	C
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

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 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6, or 7 and the output is 1 for digits 4, 5, 6 or 7. These conditions can be expressed by the following output Boolean

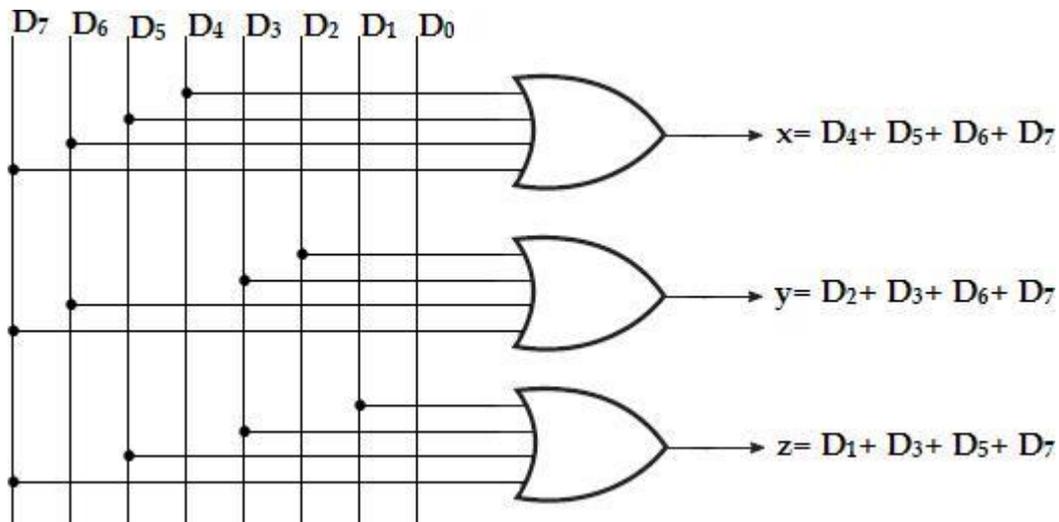
$$\text{functions: } z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

The encoder can be implemented with three OR gates. The encoder defined in the below table, 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 eg., if D_3 and D_6 are 1 simultaneously, the output of the encoder may be 111. This does not represent either D_6 or D_3 . To resolve this problem, 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 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 .



Octal-to-Binary Encoder

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

Priority Encoder:

A priority encoder is an encoder circuit that includes the priority function. In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

In addition to the two outputs x and y , the circuit has a third output, V (valid bit indicator). It 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 higher the subscript number, higher the priority of the input. Input D_3 , has the highest priority. So, regardless of the values of the other inputs, when D_3 is 1, the output for xy is 11.

D_2 has the next priority level. The output is 10, if $D_2= 1$ provided $D_3= 0$. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Truth table:

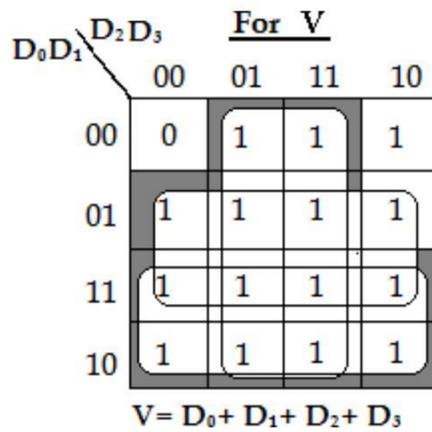
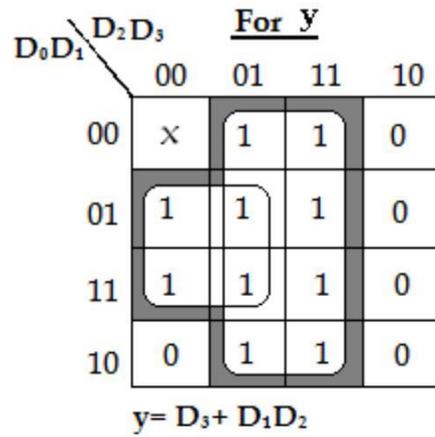
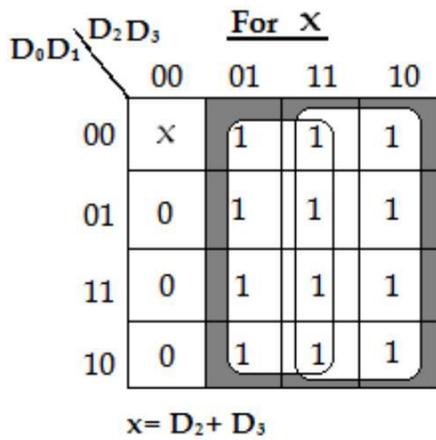
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

Although the above table has only five rows, when each don't care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the third row in the table with X100 represents minterms 0100 and 1100. The don't care condition is replaced by 0 and 1 as shown in the table below.

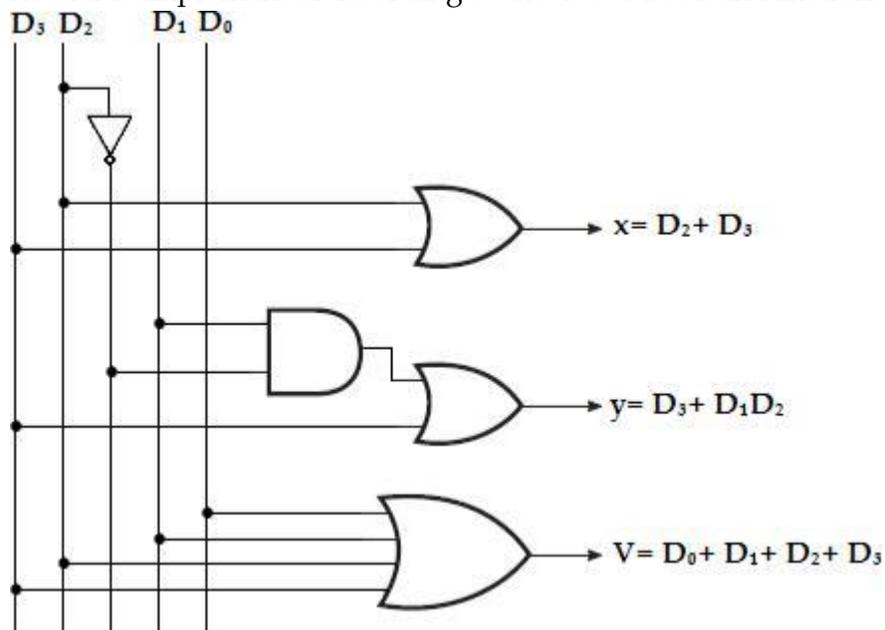
Modified Truth table:

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
0	1	0	0	0	1	1
1	1	0	0			
0	0	1	0	1	0	1
0	1	1	0			
1	0	1	0			
1	1	1	0			
0	0	0	1	1	1	1
0	0	1	1			
0	1	0	1			
0	1	1	1			
1	0	0	1			
1	0	1	1			
1	1	0	1			
1	1	1	1			

K-map Simplification:



The priority encoder is implemented according to the above Boolean functions.

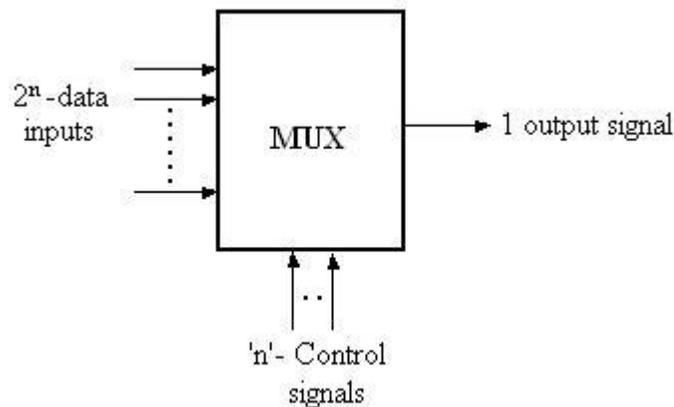


4- Input Priority Encoder

MULTIPLEXER: (Data Selector)

A *multiplexer* or *MUX*, is a combinational circuit with more than one input line, one output line and more than one selection line. A multiplexer selects binary information present from one of many input lines, depending upon the logic status of the selection inputs, and routes it to the output line. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected. The multiplexer is often labeled as MUX in block diagrams.

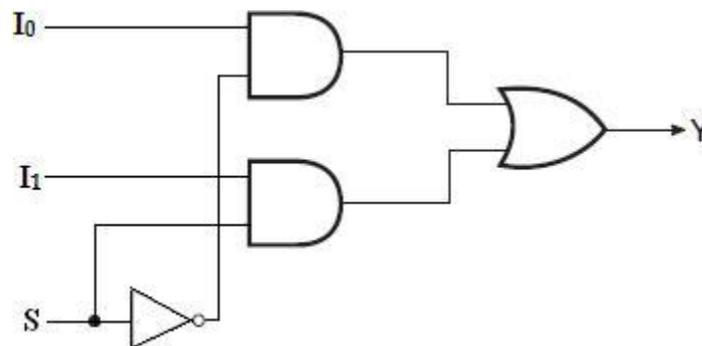
A multiplexer is also called a **data selector**, since it selects one of many inputs and steers the binary information to the output line.



Block diagram of Multiplexer

2-to-1- line Multiplexer:

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.



Logic diagram

The multiplexer acts like an electronic switch that selects one of the two sources.

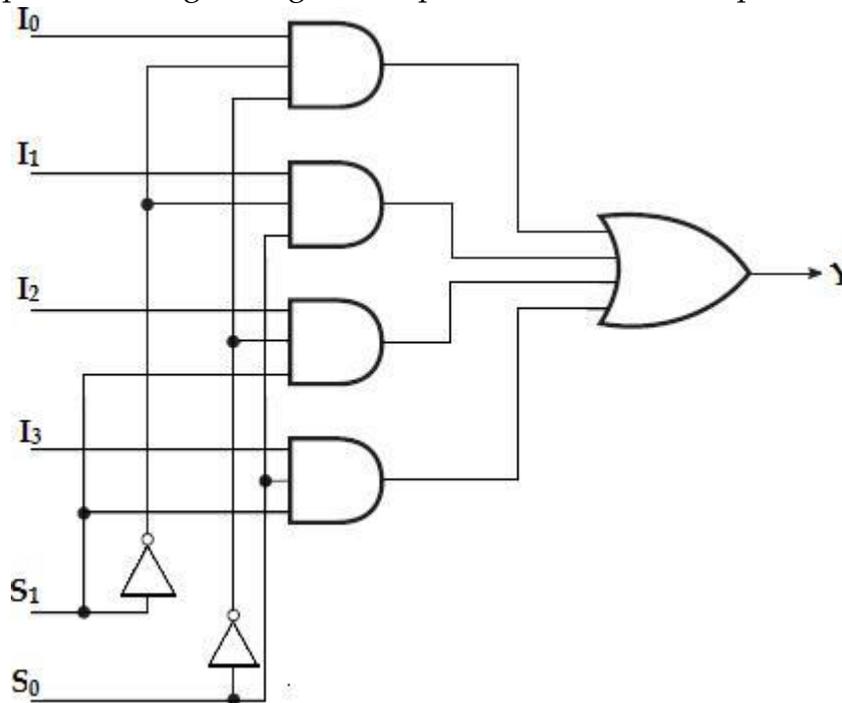
Truth table:

S	Y
0	I_0
1	I_1

4-to-1-line Multiplexer:

A 4-to-1-line multiplexer has four (2^n) input lines, two (n) select lines and one output line. It is the multiplexer consisting of four input channels and information of one of the channels can be selected and transmitted to an output line according to the select inputs combinations. Selection of one of the four input channel is possible by two selection inputs.

Each of the four inputs I_0 through I_3 , is applied to one input of AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gate are applied to a single OR gate that provides the 1-line output.



4-to-1-Line Multiplexer

Function table:

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

To demonstrate the circuit operation, 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 OR output is now equal to the value of I_2 , providing a path from the selected input to the output.

The data output is equal to I_0 only if $S_1=0$ and $S_0=0$; $Y= I_0S_1'S_0'$.

The data output is equal to I_1 only if $S_1=0$ and $S_0=1$; $Y= I_1S_1'S_0$.

The data output is equal to I_2 only if $S_1=1$ and $S_0=0$; $Y= I_2S_1S_0'$.

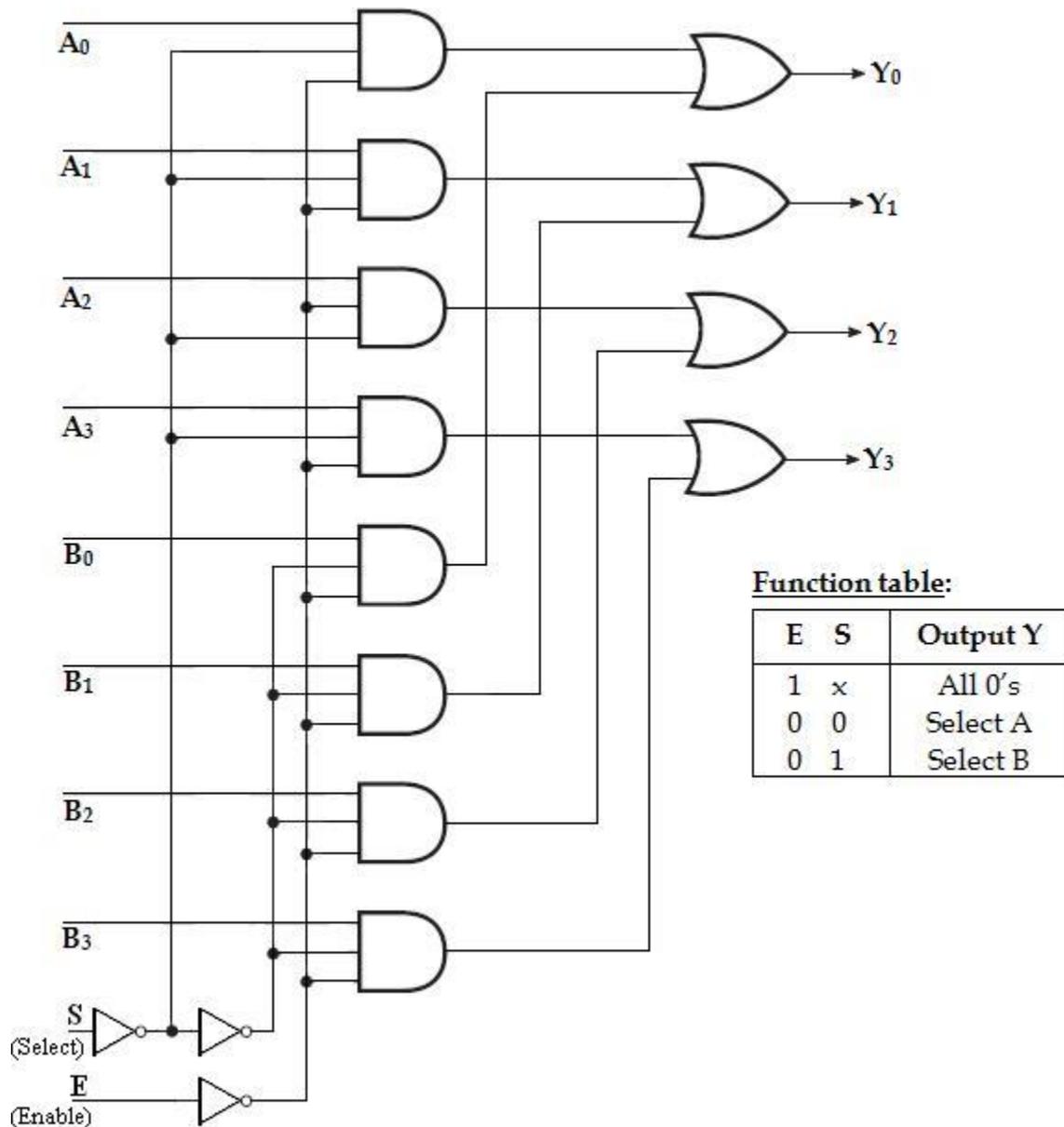
The data output is equal to I_3 only if $S_1=1$ and $S_0=1$; $Y= I_3S_1S_0$.

When these terms are ORed, the total expression for the data output is,

$$Y= I_0S_1'S_0'+ I_1S_1'S_0 + I_2S_1S_0'+ I_3S_1S_0.$$

As in decoder, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

Quadruple 2-to-1 Line Multiplexer:



Function table:

E	S	Output Y
1	x	All 0's
0	0	Select A
0	1	Select B

This circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either A_0 or B_0 . Similarly, output Y_1 may have the value of A_1 or B_1 , and so on. Input selection line, S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal operation.

Although the circuit contains four 2-to-1-Line multiplexers, it is viewed as a circuit that selects one of two 4-bit sets of data lines. The unit is enabled when $E=0$. Then if $S=0$, the four A inputs have a path to the four outputs. On the other hand, if $S=1$, the four B inputs are applied to the outputs. The outputs have all 0's when $E=1$, regardless of the value of S .

Application:

The multiplexer is a very useful MSI function and has various ranges of applications in data communication. Signal routing and data communication are the important applications of a multiplexer. It is used for connecting two or more sources to guide to a single destination among computer units and it is useful for constructing a common bus system. One of the general properties of a multiplexer is that Boolean functions can be implemented by this device.

Implementation of Boolean Function using MUX:

Any Boolean or logical expression can be easily implemented using a multiplexer. If a Boolean expression has $(n+1)$ variables, then n of these variables can be connected to the select lines of the multiplexer. The remaining single variable along with constants 1 and 0 is used as the input of the multiplexer. For example, if C is the single variable, then the inputs of the multiplexers are $C, C', 1$ and 0 . By this method any logical expression can be implemented.

In general, a Boolean expression of $(n+1)$ variables can be implemented using a multiplexer with 2^n inputs.

1. Implement the following boolean function using 4: 1 multiplexer,

$$F(A, B, C) = \sum m(1, 3, 5, 6).$$

Solution:

Variables, $n=3$ (A, B, C)

Select lines= $n-1 = 2$ (S_1, S_0)

2^{n-1} to MUX i.e., 2^2 to 1 = 4 to 1 MUX

Input lines= $2^{n-1} = 2^2 = 4$ (D_0, D_1, D_2, D_3)

Implementation table:

Apply variables A and B to the select lines. The procedures for implementing the function are:

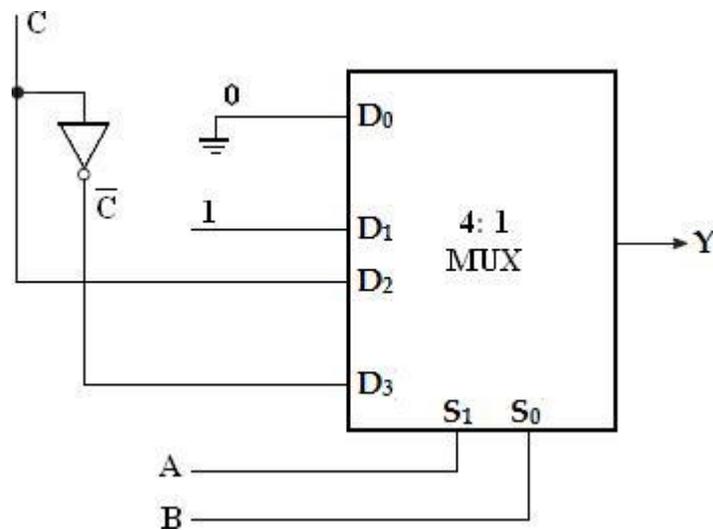
- i. List the input of the multiplexer
- ii. List under them all the minterms in two rows as shown below.

The first half of the minterms is associated with A' and the second half with A . The given function is implemented by circling the minterms of the function and applying the following rules to find the values for the inputs of the multiplexer.

1. If both the minterms in the column are not circled, apply 0 to the corresponding input.
2. If both the minterms in the column are circled, apply 1 to the corresponding input.
3. If the bottom minterm is circled and the top is not circled, apply C to the input.
4. If the top minterm is circled and the bottom is not circled, apply C' to the input.

	D ₀	D ₁	D ₂	D ₃
\bar{C}	0	1	2	3
C	4	5	6	7
	0	1	C	\bar{C}

Multiplexer Implementation:



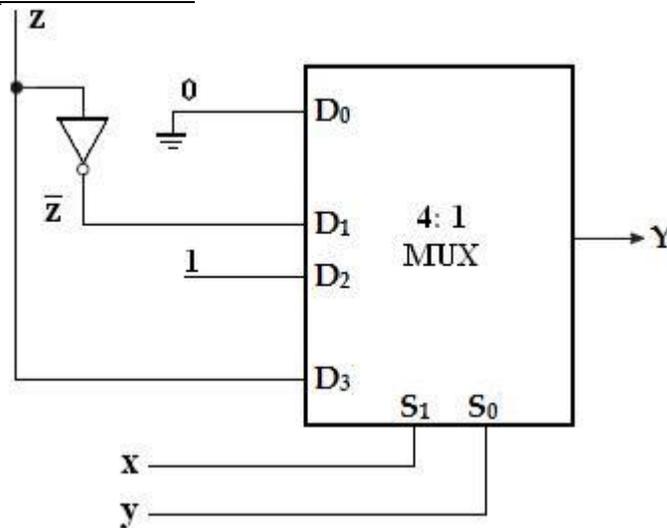
2. $F(x, y, z) = \sum m(1, 2, 6, 7)$

Solution:

Implementation table:

	D ₀	D ₁	D ₂	D ₃
\bar{z}	0	1	2	3
z	4	5	6	7
0	\bar{z}	1	z	

Multiplexer Implementation:



3. $F(A, B, C) = \sum m(1, 2, 4, 5)$

Solution:

Variables, $n=3$ (A, B, C)

Select lines = $n-1 = 2$ (S₁, S₀)

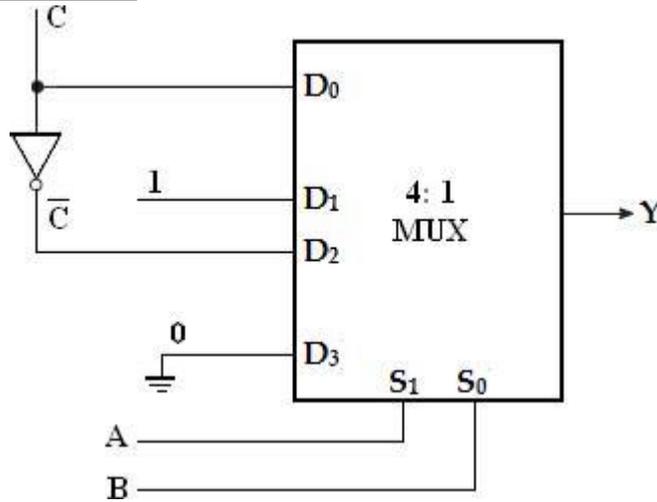
2^{n-1} to MUX i.e., 2 to 1 = 4 to 1 MUX

Input lines = $2^{n-1} = 2^2 = 4$ (D₀, D₁, D₂, D₃)

Implementation table:

	D ₀	D ₁	D ₂	D ₃
\bar{C}	0	1	2	3
C	4	5	6	7
C	1	\bar{C}	0	

Multiplexer Implementation:



4. $F(P, Q, R, S) = \sum m(0, 1, 3, 4, 8, 9, 15)$

Solution:

Variables, $n = 4$ (P, Q, R, S)

Select lines = $n - 1 = 3$ (S_2, S_1, S_0)

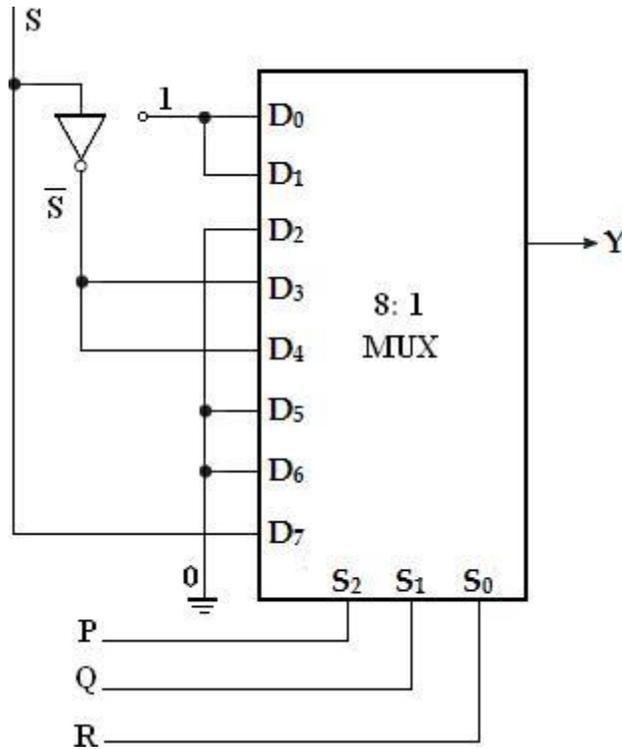
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines = $2^{n-1} = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

Implementation table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{S}	0	1	2	3	4	5	6	7
S	8	9	10	11	12	13	14	15
	1	1	0	\bar{S}	\bar{S}	0	0	S

Multiplexer Implementation:



5. Implement the Boolean function using 8: 1 and also using 4:1 multiplexer
 $F(A, B, C, D) = \sum m(0, 1, 2, 4, 6, 9, 12, 14)$

Solution:

Variables, $n=4$ (A, B, C, D)

Select lines= $n-1 = 3$ (S_2, S_1, S_0)

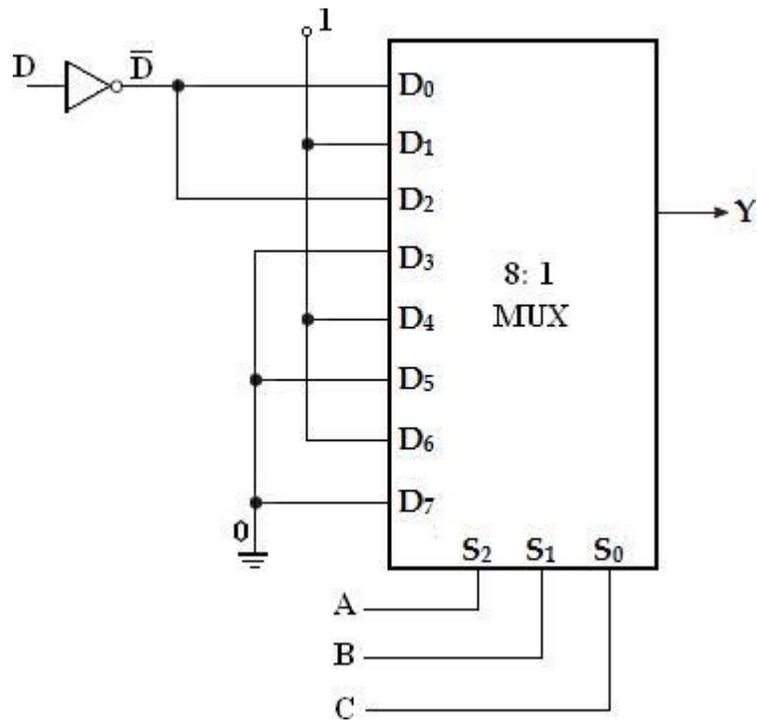
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

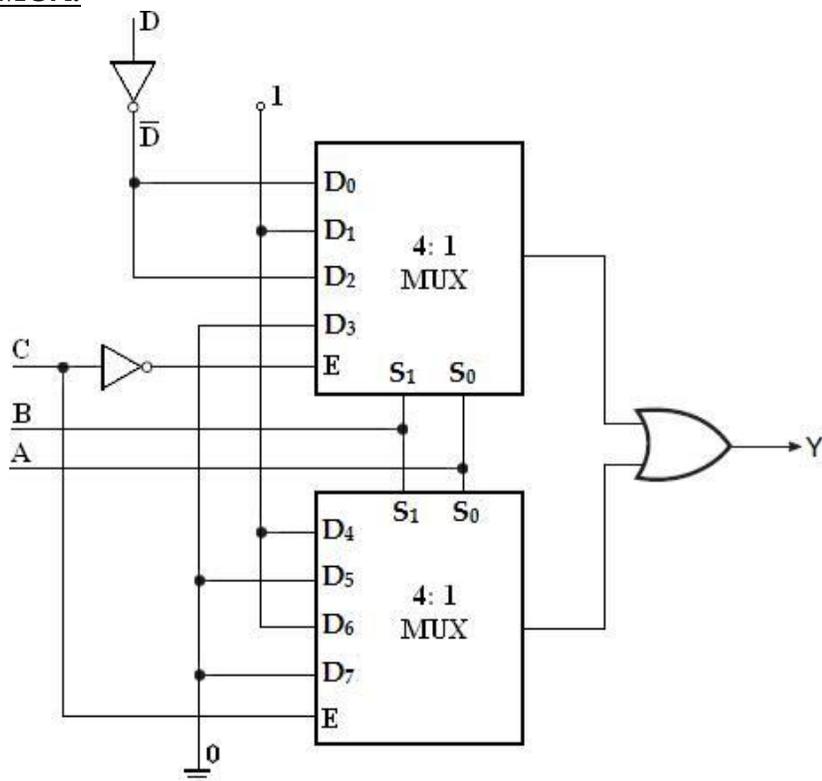
Implementation table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	\bar{D}	1	\bar{D}	0	1	0	1	0

Multiplexer Implementation (Using 8: 1 MUX):



Using 4: 1 MUX:



6. $F(A, B, C, D) = \sum m(1, 3, 4, 11, 12, 13, 14, 15)$

Solution:

Variables, $n=4$ (A, B, C, D)

Select lines= $n-1 = 3$ (S_2, S_1, S_0)

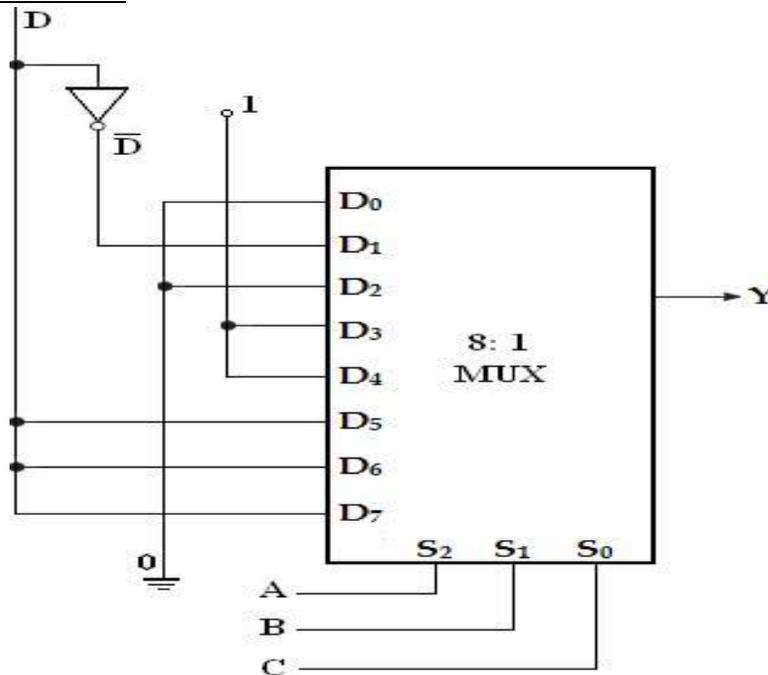
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines= $2^n = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

Implementation table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	0	\bar{D}	0	1	1	D	D	D

Multiplexer Implementation:



7. Implement the Boolean function using 8: 1 multiplexer.

$$F(A, B, C, D) = A'BD' + ACD + B'CD + A'C'D.$$

Solution:

Convert into standard SOP form,

$$= A'BD' (C'+C) + ACD (B'+B) + B'CD (A'+A) + A'C'D (B'+B)$$

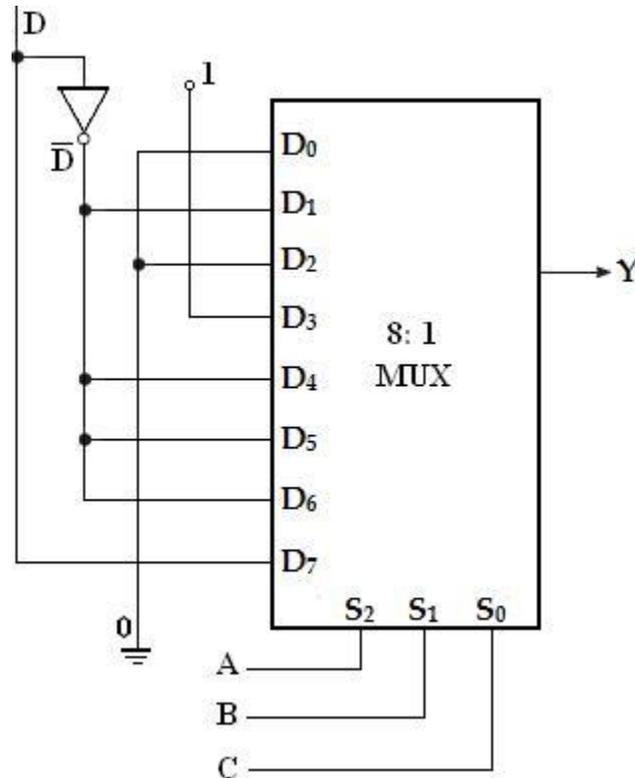
$$= A'BC'D' + A'BCD' + \underline{AB'CD} + ABCD + A'B'CD + \underline{AB'CD} + A'B'C'D + A'BC'D$$

$$\begin{aligned}
 &= A'BC'D' + A'BCD' + AB'CD + ABCD + A'B'CD + A'B'C'D + A'BC'D \\
 &= m_4 + m_6 + m_{11} + m_{15} + m_3 + m_1 + m_5 \\
 &= \sum m(1, 3, 4, 5, 6, 11, 15)
 \end{aligned}$$

Implementation table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	0	\bar{D}	0	1	\bar{D}	\bar{D}	\bar{D}	D

Multiplexer Implementation:



8. Implement the Boolean function using 8: 1 multiplexer.

$$F(A, B, C, D) = AB'D + A'C'D + B'CD' + AC'D.$$

Solution:

Convert into standard SOP form,

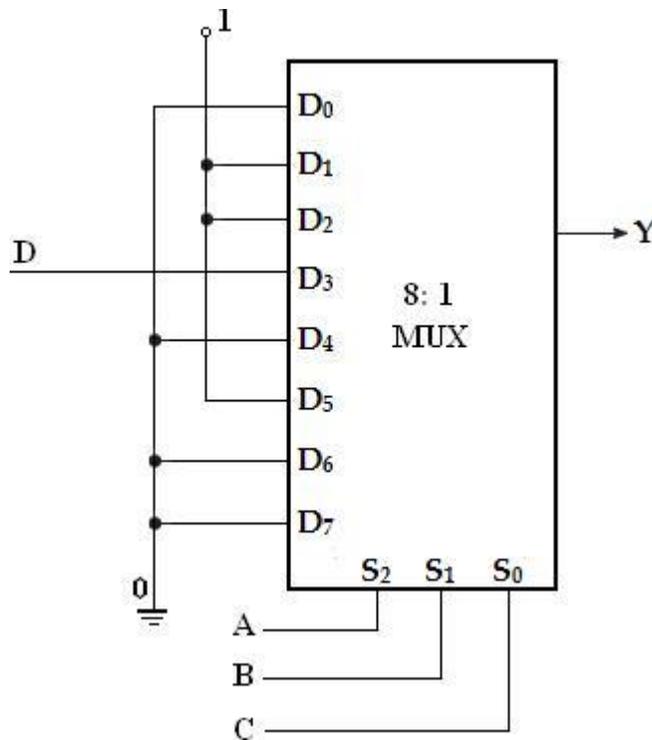
$$= AB'D(C'+C) + A'C'D(B'+B) + B'CD'(A'+A) + AC'D(B'+B)$$

$$\begin{aligned}
&= \underline{AB'CD} + AB'CD + A'B'CD + A'BC'D + A'B'CD' + AB'CD' + \underline{AB'CD} + ABC'D \\
&= AB'CD + AB'CD + A'B'CD + A'BC'D + A'B'CD' + AB'CD' + ABC'D \\
&= m_9 + m_{11} + m_1 + m_5 + m_2 + m_{10} + m_{13} \\
&= \sum m(1, 2, 5, 9, 10, 11, 13).
\end{aligned}$$

Implementation Table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	0	1	1	D	0	1	0	0

Multiplexer Implementation:



9. Implement the Boolean function using 8: 1 and also using 4:1 multiplexer
 $F(w, x, y, z) = \sum m(1, 2, 3, 6, 7, 8, 11, 12, 14)$

Solution:

Variables, $n = 4$ (w, x, y, z)
 Select lines = $n - 1 = 3$ (S_2, S_1, S_0)

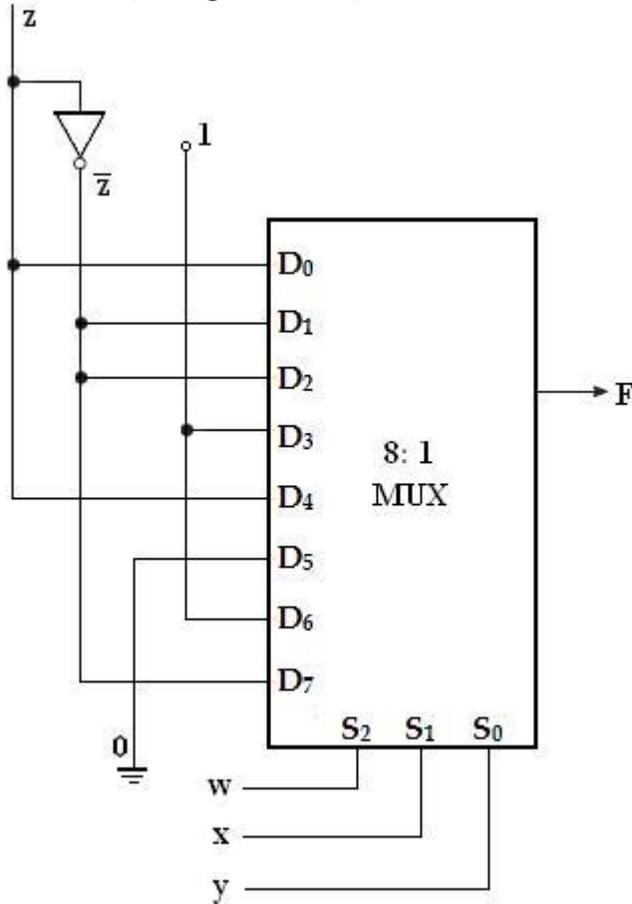
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines = $2^{n-1} = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

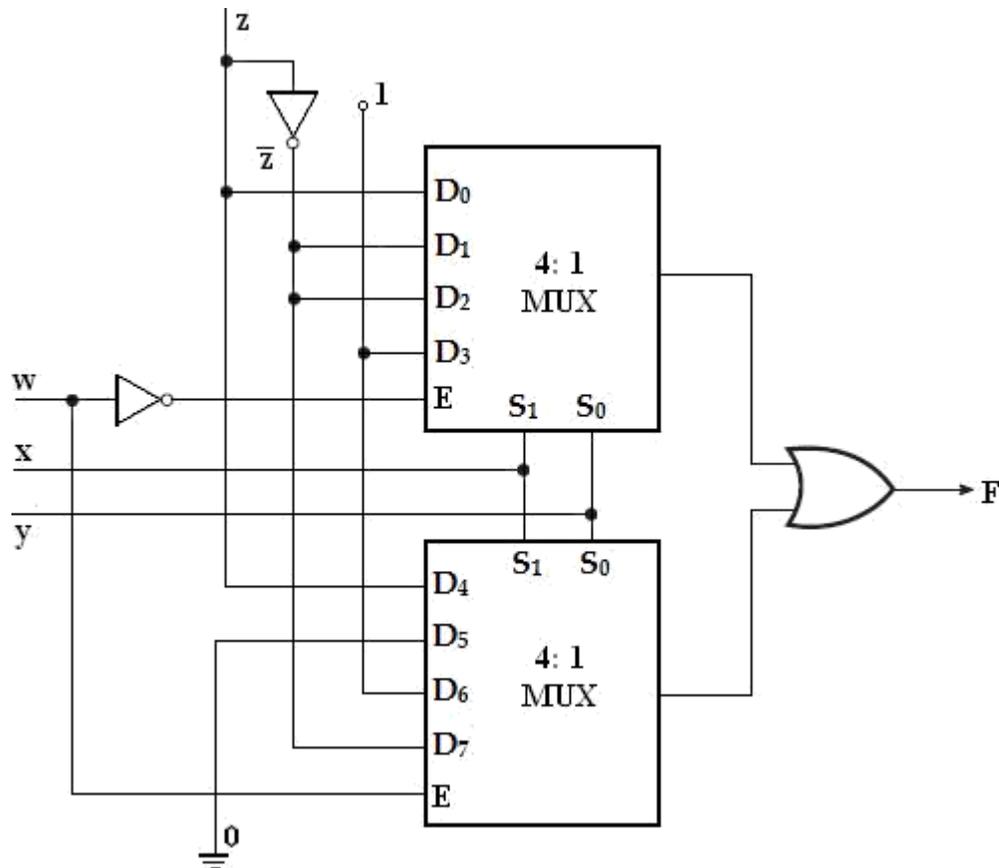
Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
\bar{z}	0	1	2	3	4	5	6	7
z	8	9	10	11	12	13	14	15
	z	\bar{z}	\bar{z}	1	z	0	1	\bar{z}

Multiplexer Implementation (Using 8:1 MUX):



(Using 4:1 MUX):



10. Implement the Boolean function using 8: 1 multiplexer

$$F(A, B, C, D) = \sum m(0, 3, 5, 8, 9, 10, 12, 14)$$

Solution:

Variables, $n=4$ (A, B, C, D)

Select lines = $n-1 = 3$ (S_2, S_1, S_0)

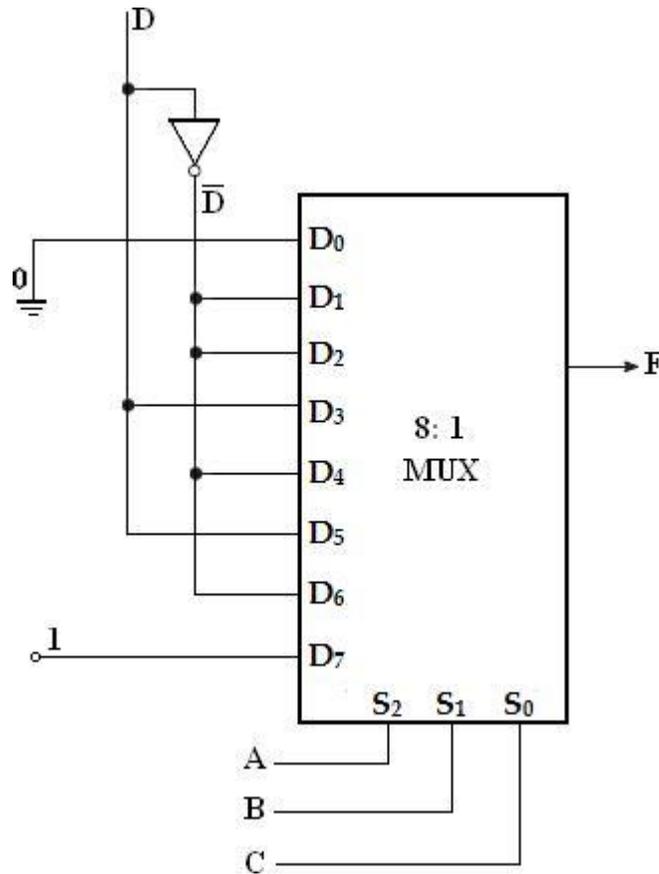
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines = $2^{n-1} = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

Implementation table:

	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	0	\bar{D}	\bar{D}	D	\bar{D}	D	\bar{D}	1

Multiplexer Implementation:



11. Implement the Boolean function using 8: 1 multiplexer

$$F(A, B, C, D) = \sum m(0, 2, 6, 10, 11, 12, 13) + d(3, 8, 14)$$

Solution:

Variables, $n=4$ (A, B, C, D)

Select lines = $n-1 = 3$ (S_2, S_1, S_0)

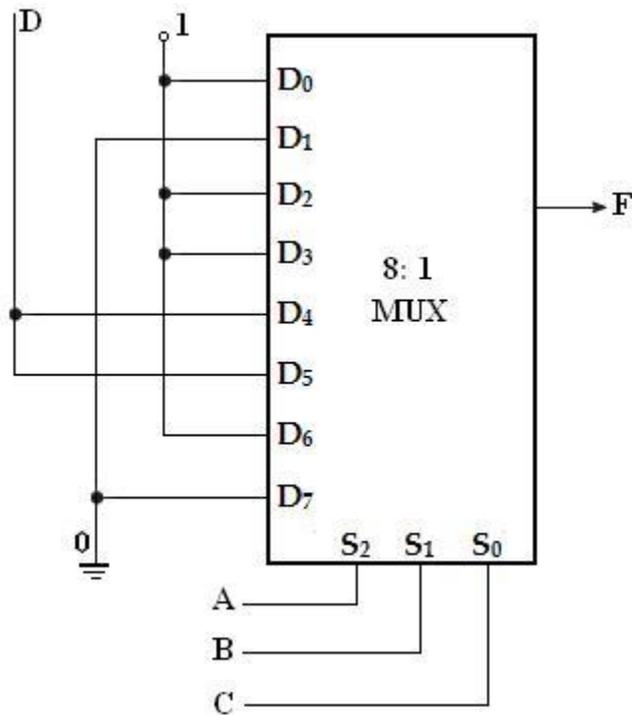
2^{n-1} to MUX i.e., 2^3 to $1 = 8$ to 1 MUX

Input lines = $2^n = 2^3 = 8$ ($D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7$)

Implementation Table:

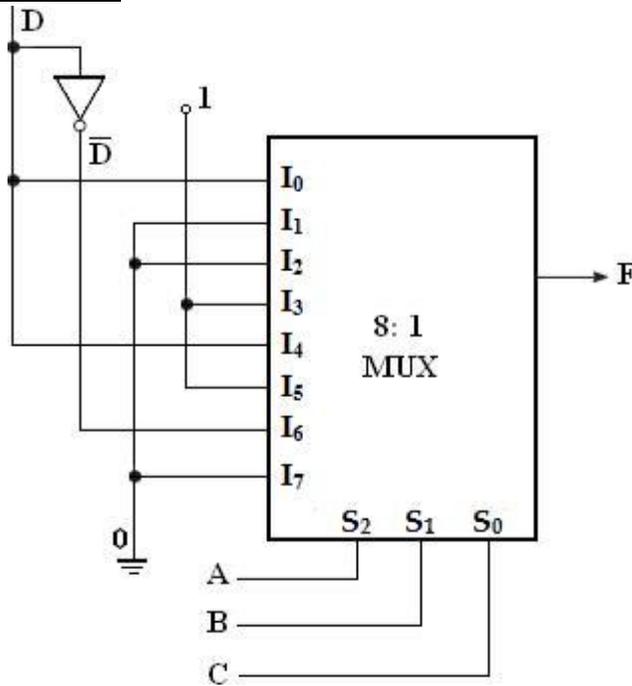
	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	1	0	1	1	D	D	1	0

Multiplexer Implementation:



12. An 8×1 multiplexer has inputs A , B and C connected to the selection inputs S_2 , S_1 , and S_0 respectively. The data inputs I_0 to I_7 are as follows
 $I_1=I_2=I_7= 0$; $I_3=I_5= 1$; $I_0=I_4= D$ and $I_6= D'$.
 Determine the Boolean function that the multiplexer implements.

Multiplexer Implementation:



Implementation table:

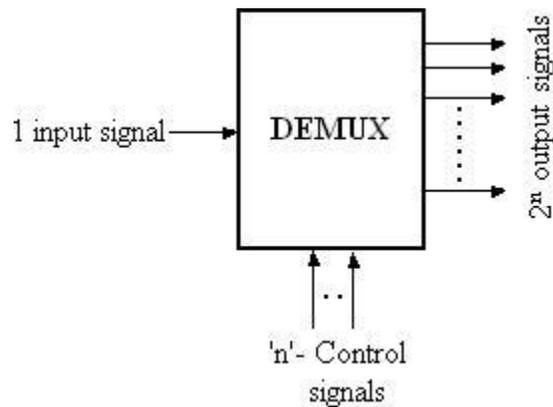
	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
\bar{D}	0	1	2	3	4	5	6	7
D	8	9	10	11	12	13	14	15
	D	0	0	1	D	1	\bar{D}	0

$F(A, B, C, D) = \sum m(3, 5, 6, 8, 11, 12, 13).$

DEMULTIPLEXER:

Demultiplex means one into many. Demultiplexing is the process of taking information from one input and transmitting the same over one of several outputs.

A demultiplexer is a combinational logic circuit that receives information on a single input and transmits the same information over one of several (2ⁿ) output lines.

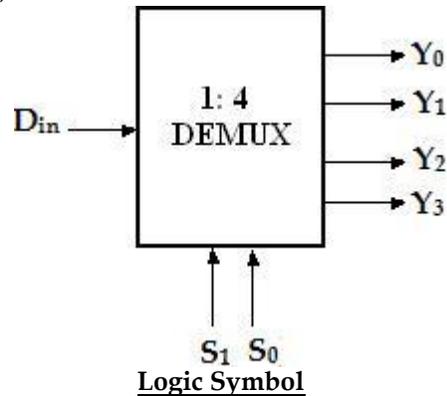


Block diagram of demultiplexer

The block diagram of a demultiplexer which is opposite to a multiplexer in its operation is shown above. The circuit has one input signal, n' select signals and 2^n output signals. The select inputs determine to which output the data input will be connected. As the serial data is changed to parallel data, i.e., the input caused to appear on one of the n output lines, the demultiplexer is also called a *data distributor* or a *serial-to-parallel converter*.

1-to-4 Demultiplexer:

A 1-to-4 demultiplexer has a single input, D_{in} , four outputs (Y_0 to Y_3) and two select inputs (S_1 and S_0).



The input variable D_{in} has a path to all four outputs, but the input information is directed to only one of the output lines. The truth table of the 1-to-4 demultiplexer is shown below.

Enable	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3
0	x	x	x	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1

Truth table of 1-to-4 demultiplexer

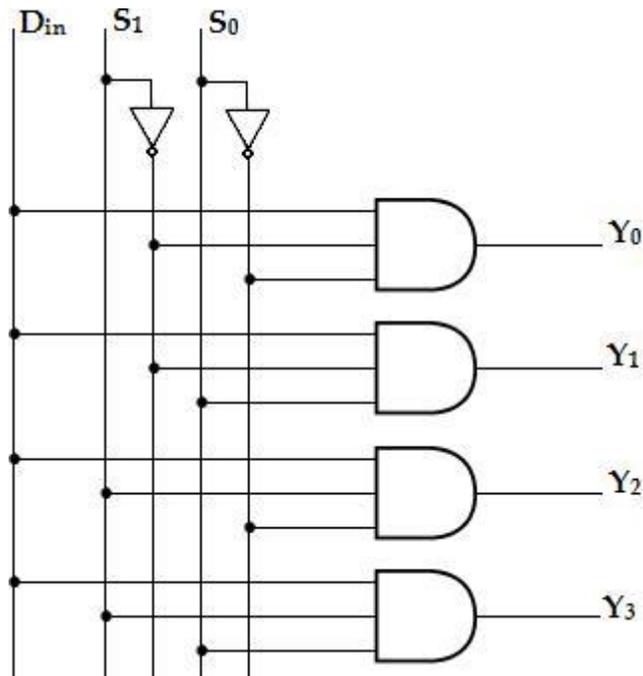
From the truth table, it is clear that the data input, D_{in} is connected to the output Y_0 , when $S_1=0$ and $S_0=0$ and the data input is connected to output Y_1 when $S_1=0$ and $S_0=1$. Similarly, the data input is connected to output Y_2 and Y_3 when $S_1=1$ and $S_0=0$ and when $S_1=1$ and $S_0=1$, respectively. Also, from the truth table, the expression for outputs can be written as follows,

$$Y_0 = S_1' S_0' D_{in}$$

$$Y_1 = S_1' S_0 D_{in}$$

$$Y_2 = S_1 S_0' D_{in}$$

$$Y_3 = S_1 S_0 D_{in}$$



Logic diagram of 1-to-4 demultiplexer

Now, using the above expressions, a 1-to-4 demultiplexer can be implemented using four 3-input AND gates and two NOT gates. Here, the input data line D_{in} , is connected to all the AND gates. The two select lines S_1 , S_0 enable only one gate at a time and the data that appears on the input line passes through the selected gate to the associated output line.

1-to-8 Demultiplexer:

A 1-to-8 demultiplexer has a single input, D_{in} , eight outputs (Y_0 to Y_7) and three select inputs (S_2 , S_1 and S_0). It distributes one input line to eight output lines based on the select inputs. The truth table of 1-to-8 demultiplexer is shown below.

D_{in}	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0

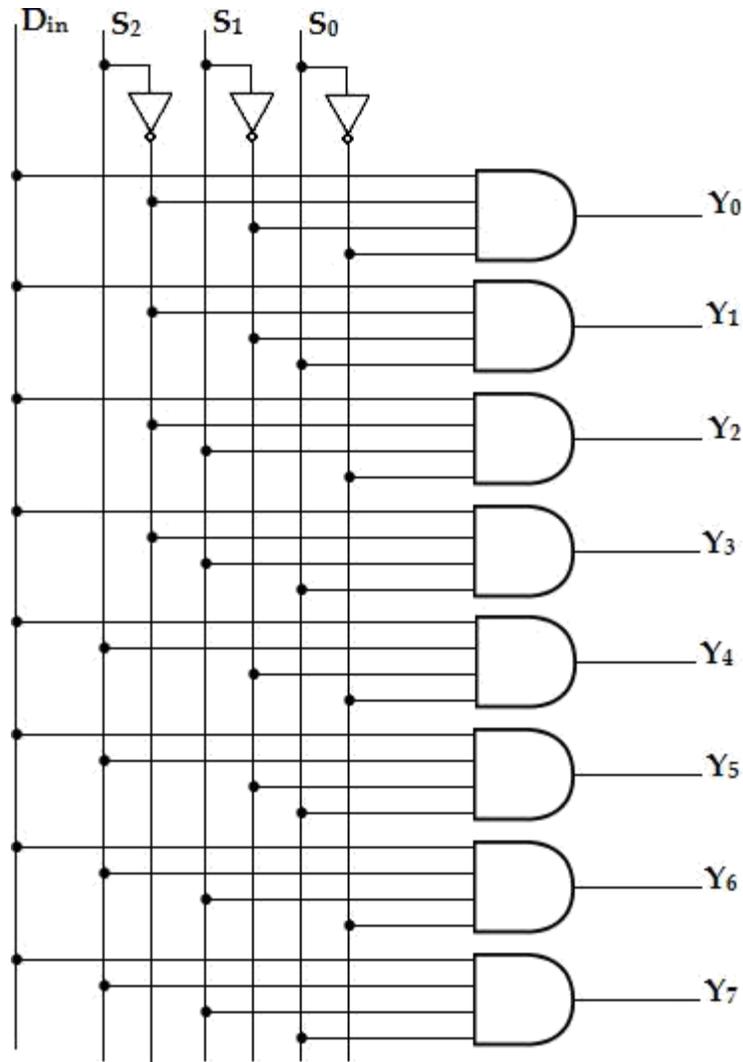
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Truth table of 1-to-8 demultiplexer

From the above truth table, it is clear that the data input is connected with one of the eight outputs based on the select inputs. Now from this truth table, the expression for eight outputs can be written as follows:

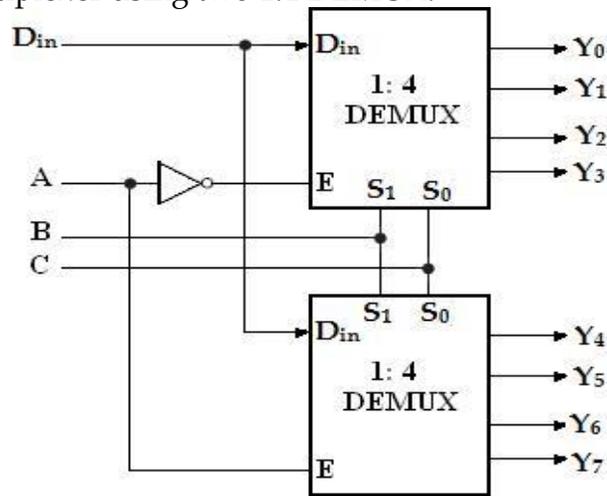
$$\begin{aligned}
 Y_0 &= S_2' S_1' S_0' D_{in} & Y_4 &= S_2 S_1' S_0' D_{in} \\
 Y_1 &= S_2' S_1' S_0 D_{in} & Y_5 &= S_2 S_1 S_0' D_{in} \\
 Y_2 &= S_2' S_1 S_0' D_{in} & Y_6 &= S_2 S_1 S_0 D_{in} \\
 Y_3 &= S_2' S_1 S_0 D_{in} & Y_7 &= S_2 S_1' S_0 D_{in}
 \end{aligned}$$

Now using the above expressions, the logic diagram of a 1-to-8 demultiplexer can be drawn as shown below. Here, the single data line, D_{in} is connected to all the eight AND gates, but only one of the eight AND gates will be enabled by the select input lines. For example, if $S_2 S_1 S_0 = 000$, then only AND gate-0 will be enabled and thereby the data input, D_{in} will appear at Y_0 . Similarly, the different combinations of the select inputs, the input D_{in} will appear at the respective output.



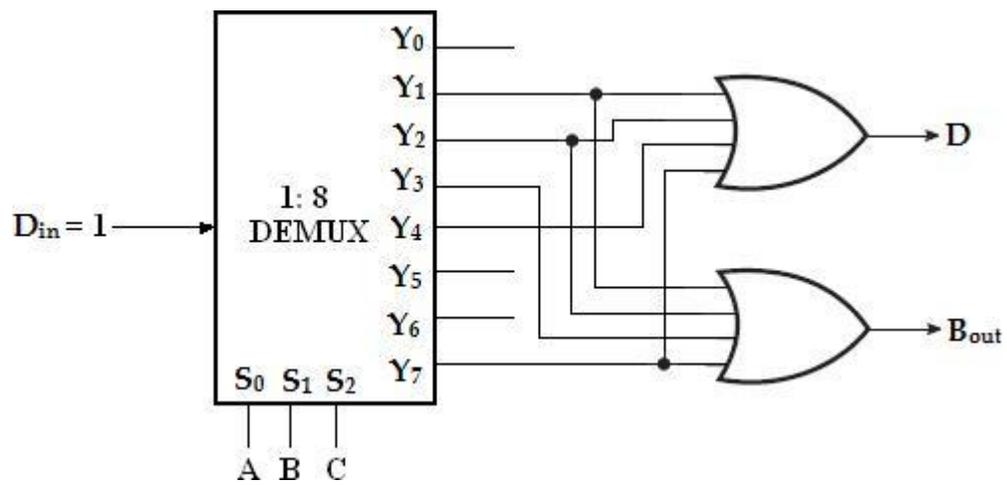
Logic diagram of 1-to-8 demultiplexer

1. Design 1:8 demultiplexer using two 1:4 DEMUX.



2. Implement full subtractor using demultiplexer.

Inputs			Outputs	
A	B	B _{in}	Difference(D)	Borrow(B _{out})
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

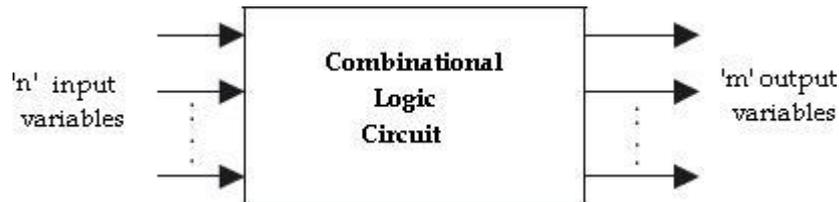


UNIT II
SYNCHRONOUS SEQUENTIAL LOGIC

Introduction to Sequential Circuits – Flip-Flops – operation and excitation tables, Triggering of FF, Analysis and design of clocked sequential circuits – Design – Moore/Mealy models, state minimization, state assignment, circuit implementation - Registers – Counters.

INTRODUCTION

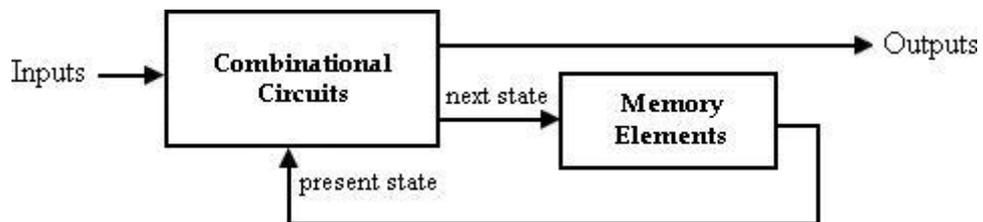
In *combinational logic circuits*, the outputs at any instant of time depend only on the input signals present at that time. For a change in input, the output occurs immediately.



Combinational Circuit- Block Diagram

In *sequential logic circuits*, it consists of combinational circuits to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information either 1 or 0.

The information stored in the memory elements at any given time defines the present state of the sequential circuit. The present state and the external circuit determine the output and the next state of sequential circuits.



Sequential Circuit- Block Diagram

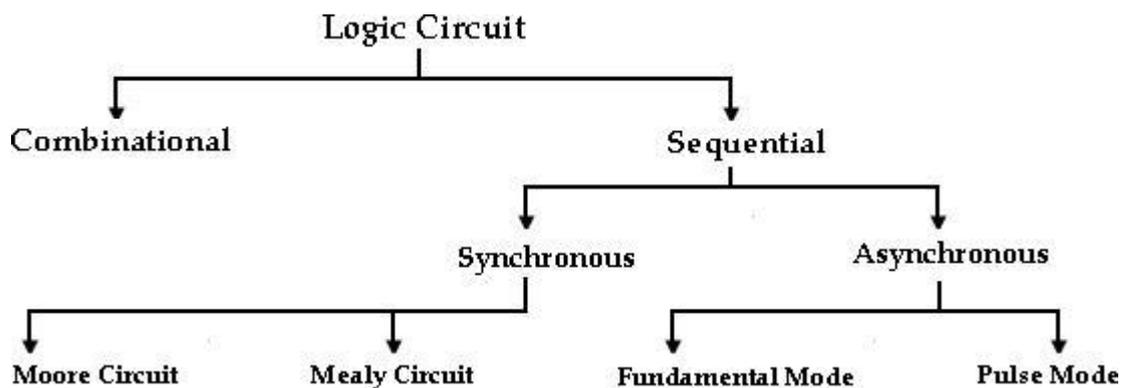
Thus in sequential circuits, the output variables depend not only on the present input variables but also on the past history of input variables.

The rotary channel selected knob on an old-fashioned TV is like a combinational. Its output selects a channel based only on its current input – the position of the knob. The channel-up and channel-down push buttons on a TV is like a sequential circuit. The channel selection depends on the past sequence of up/ down pushes.

The comparison between combinational and sequential circuits is given in table below.

S.No	Combinational logic	Sequential logic
1	The output variable, at all times depends on the combination of input variables.	The output variable depends not only on the present input but also depend upon the past history of inputs.
2	Memory unit is not required	Memory unit is required to store the past history of input variables.
3	Faster in speed	Slower than combinational circuits.
4	Easy to design	Comparatively harder to design.
5	Eg. Parallel adder	Eg. Serial adder

3.2 Classification of Logic Circuits



The sequential circuits can be classified depending on the timing of their signals:

- x Synchronous sequential circuits
- x Asynchronous sequential circuits.

In synchronous sequential circuits, signals can affect the memory elements only at discrete instants of time. In asynchronous sequential circuits change in input signals can affect memory element at any instant of time. The memory elements used in both circuits are Flip-Flops, which are capable of storing 1-bit information.

S.No	Synchronous sequential circuits	Asynchronous sequential circuits
1	Memory elements are clocked Flip-Flops	Memory elements are either unclocked Flip-Flops or time delay elements.
2	The change in input signals can affect memory element upon activation of clock signal.	The change in input signals can affect memory element at any instant of time.
3	The maximum operating speed of clock depends on time delays involved.	Because of the absence of clock, it can operate faster than synchronous circuits.
4	Easier to design	More difficult to design

3.3 LATCHES:

Latches and Flip-Flops are the basic building blocks of the most sequential circuits. Latches are used for a sequential device that checks all of its inputs continuously and changes its outputs accordingly at any time independent of clocking signal. Enable signal is provided with the latch. When enable signal is active output changes occur as the input changes. But when enable signal is not activated input changes do not affect the output.

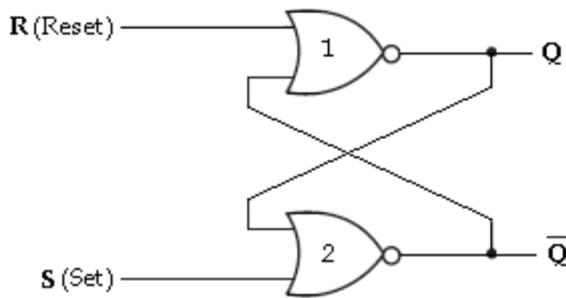
Flip-Flop is used for a sequential device that normally samples its inputs and changes its outputs only at times determined by clocking signal.

3.3.1 SR Latch:

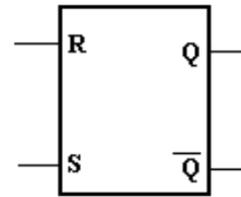
The simplest type of latch is the set-reset (SR) latch. It can be constructed from either two NOR gates or two NAND gates.

SR latch using NOR gates:

The two NOR gates are cross-coupled so that the output of NOR gate 1 is connected to one of the inputs of NOR gate 2 and vice versa. The latch has two outputs Q and Q' and two inputs, set and reset.



SR latch using NOR gates



Logic Symbol

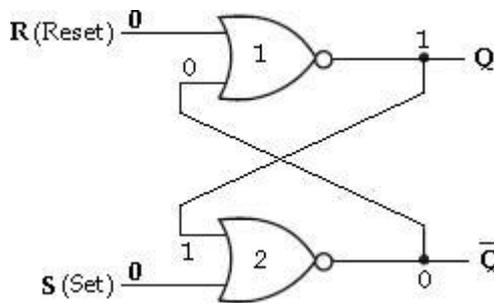
Before going to analyse the SR latch, we recall that a logic 1 at any input of a NOR gate forces its output to a logic 0. Let us understand the operation of this circuit for various input/ output possibilities.

Case 1: S= 0 and R= 0

Initially, Q= 1 and Q'= 0

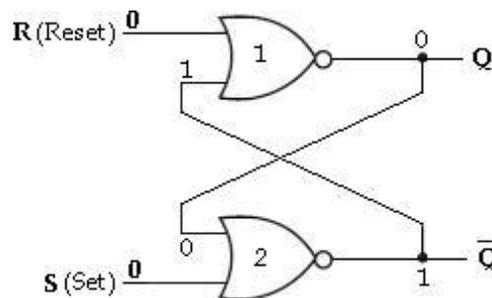
Let us assume that initially Q=1 and Q'=0. With Q'=0, both inputs to NOR gate 1 are at logic 0. So, its output, Q is at logic 1. With Q=1, one input of NOR gate 2 is at logic

1. Hence its output, Q' is at logic 0. This shows that when S and R both are low, the output does not change.



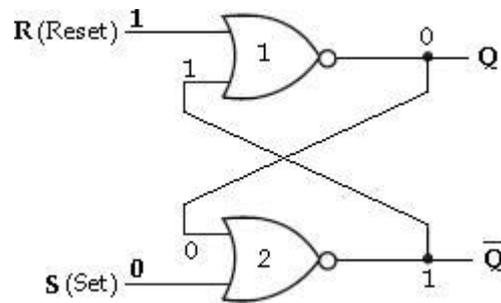
Initially, Q= 0 and Q'= 1

With Q'=1, one input of NOR gate 1 is at logic 1, hence its output, Q is at logic 0. With Q=0, both inputs to NOR gate 2 are at logic 0. So, its output Q' is at logic 1. In this case also there is no change in the output state.



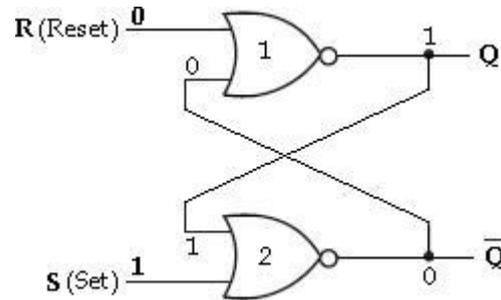
Case 2: S= 0 and R= 1

In this case, R input of the NOR gate 1 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 2 are now at logic 0. So that its output, Q' is at logic 1.



Case 3: S= 1 and R= 0

In this case, S input of the NOR gate 2 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 1 are now at logic 0. So that its output, Q is at logic 1.



Case 4: S= 1 and R= 1

When R and S both are at logic 1, they force the outputs of both NOR gates to the low state, i.e., (Q=0 and Q'=0). So, we call this an indeterminate or prohibited state, and represent this condition in the truth table as an asterisk (*). This condition also violates the basic definition of a latch that requires Q to be complement of Q'. Thus in normal operation this condition must be avoided by making sure that 1's are not applied to both the inputs simultaneously.

We can summarize the operation of SR latch as follows:

- x When S= 0 and R= 0, the output, Q_{n+1} remains in its present state, Q_n .
- x When S= 0 and R= 1, the latch is reset to 0.
- x When S= 1 and R= 0, the latch is set to 1.
- x When S= 1 and R= 1, the output of both gates will produce 0.

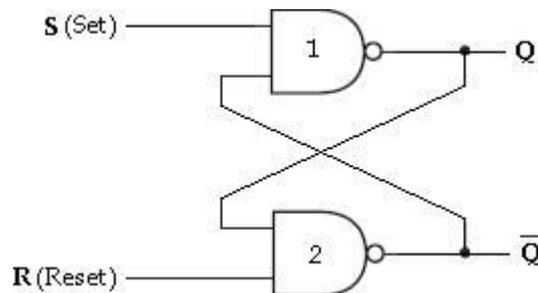
i.e., $Q_{n+1} = Q_{n+1}' = 0$.

The truth table of NOR based SR latch is shown below.

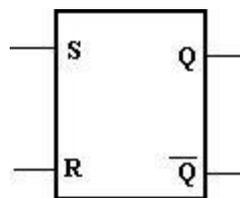
S	R	Q_n	Q_{n+1}	State
0	0	0	0	No Change (NC)
0	0	1	1	
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	x	Indeterminate *
1	1	1	x	

SR latch using NAND gates:

The SR latch can also be implemented using NAND gates. The inputs of this Latch are S and R. To understand how this circuit functions, recall that a low on any input to a NAND gate forces its output high.



SR latch using NAND gates



Logic Symbol

We can summarize the operation of SR latch as follows:

- x When $S=0$ and $R=0$, the output of both gates will produce 0.
i.e., $Q_{n+1} = Q_n = 0$.
- x When $S=0$ and $R=1$, the latch is reset to 0.
- x When $S=1$ and $R=0$, the latch is set to 1.
- x When $S=1$ and $R=1$, the output, Q_{n+1} remains in its present state, Q_n .

The truth table of NAND based SR latch is shown below.

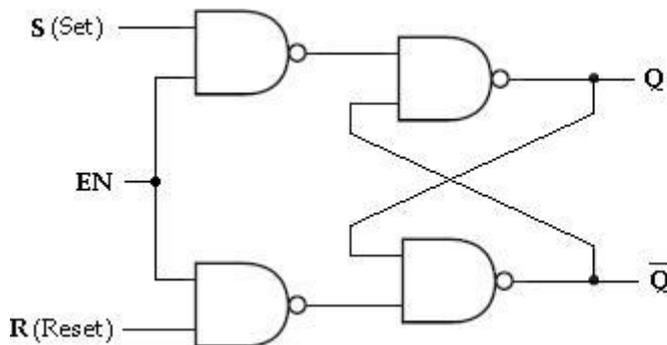
S	R	Q_n	Q_{n+1}	State
0	0	0	x	Indeterminate *
0	0	1	x	
0	1	0	1	Set
0	1	1	1	
1	0	0	0	Reset
1	0	1	0	
1	1	0	0	No Change (NC)
1	1	1	1	

Gated SR Latch:

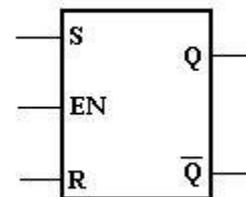
In the SR latch, the output changes occur immediately after the input changes i.e, the latch is sensitive to its S and R inputs all the time.

A latch that is sensitive to the inputs only when an enable input is active. Such a latch with enable input is known as gated SR latch.

- x The circuit behaves like SR latch when $EN=1$. It retains its previous state when $EN=0$



SR Latch with enable input using NAND gates



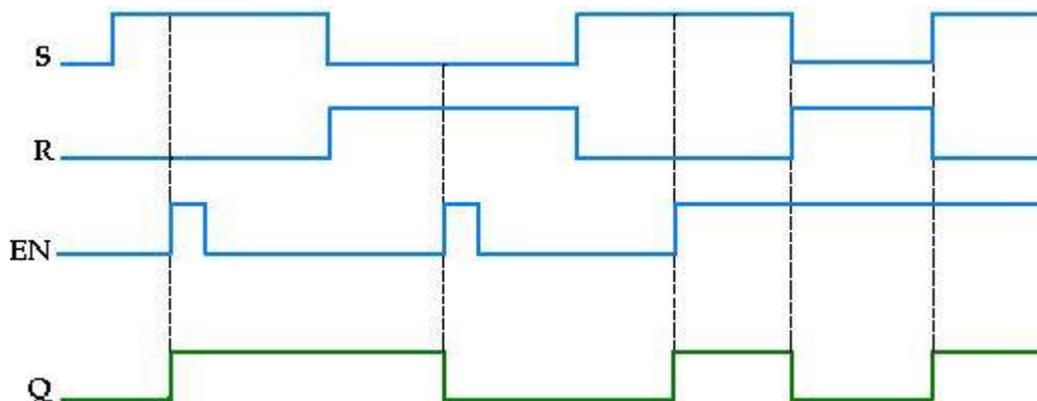
Logic Symbol

The truth table of gated SR latch is show below.

EN	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	

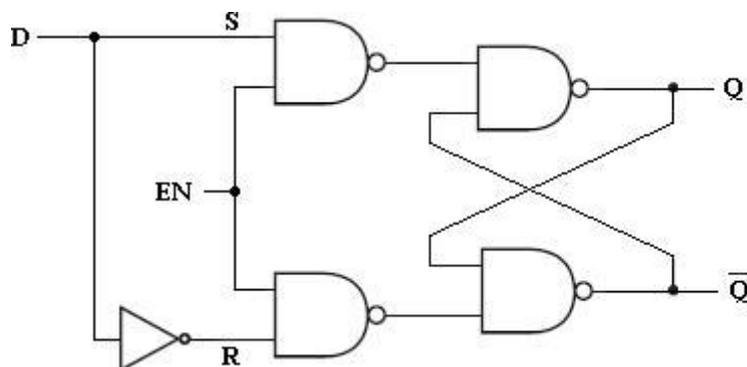
1	1	0	0	1	Set
1	1	0	1	1	
1	1	1	0	x	Indeterminate *
1	1	1	1	x	
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

When S is HIGH and R is LOW, a HIGH on the EN input sets the latch. When S is LOW and R is HIGH, a HIGH on the EN input resets the latch.

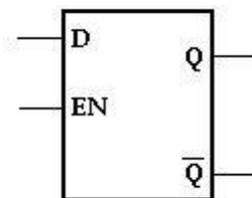


3.3.2 D Latch

In SR latch, when both inputs are same (00 or 11), the output either does not change or it is invalid. In many practical applications, these input conditions are not required. These input conditions can be avoided by making them complement of each other. This modified SR latch is known as **D latch**.



D Latch



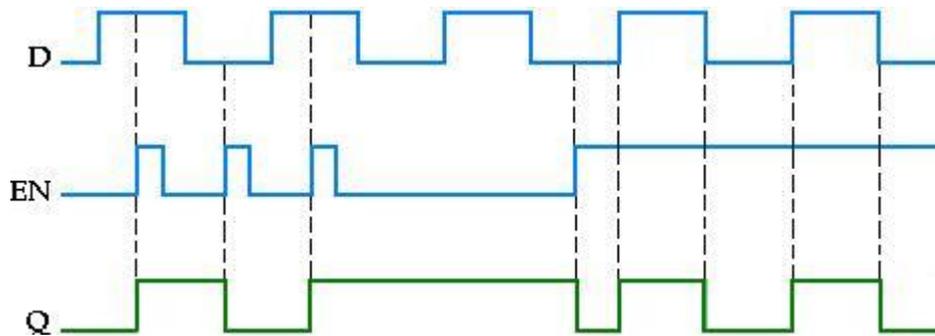
Logic Symbol

As shown in the figure, D input goes directly to the S input, and its complement is applied to the R input. Therefore, only two input conditions exist, either $S=0$ and $R=1$ or $S=1$ and $R=0$. The truth table for D latch is shown below.

EN	D	Q_n	Q_{n+1}	State
1	0	x	0	Reset
1	1	x	1	Set
0	x	x	Q_n	No Change (NC)

As shown in the truth table, the Q output follows the D input. For this reason, D latch is called *transparent latch*.

When D is HIGH and EN is HIGH, Q goes HIGH. When D is LOW and EN is HIGH, Q goes LOW. When EN is LOW, the state of the latch is not affected by the D input.

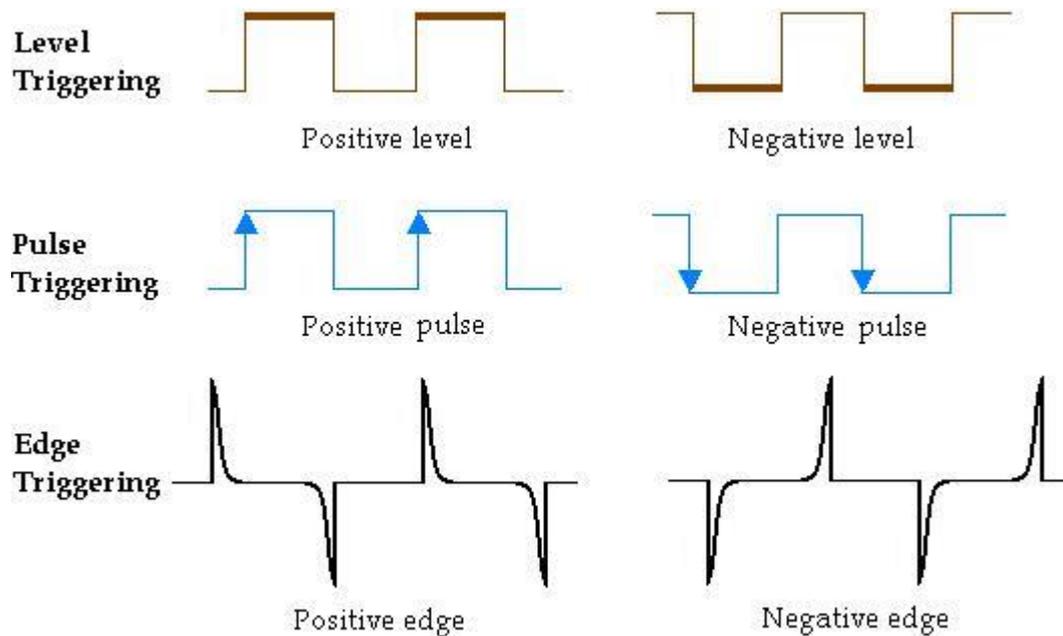


3.4 TRIGGERING OF FLIP-FLOPS

The state of a Flip-Flop is switched by a momentary change in the input signal. This momentary change is called a trigger and the transition it causes is said to trigger the Flip-Flop. Clocked Flip-Flops are triggered by pulses. A clock pulse starts from an initial value of 0, goes momentarily to 1 and after a short time, returns to its initial 0 value.

Latches are controlled by enable signal, and they are level triggered, either positive level triggered or negative level triggered. The output is free to change according to the S and R input values, when active level is maintained at the enable input.

Flip-Flops are different from latches. Flip-Flops are pulse or clock edge triggered instead of level triggered.



3.5 EDGE TRIGGERED FLIP-FLOPS

Flip-Flops are synchronous bistable devices (has two outputs Q and Q'). In this case, the term synchronous means that the output changes state only at a specified point on the triggering input called the clock (CLK), i.e., changes in the output occur in synchronization with the clock.

An *edge-triggered Flip-Flop* changes state either at the positive edge (rising edge) or at the negative edge (falling edge) of the clock pulse and is sensitive to its inputs only at this transition of the clock. The different types of edge-triggered Flip-Flops are –

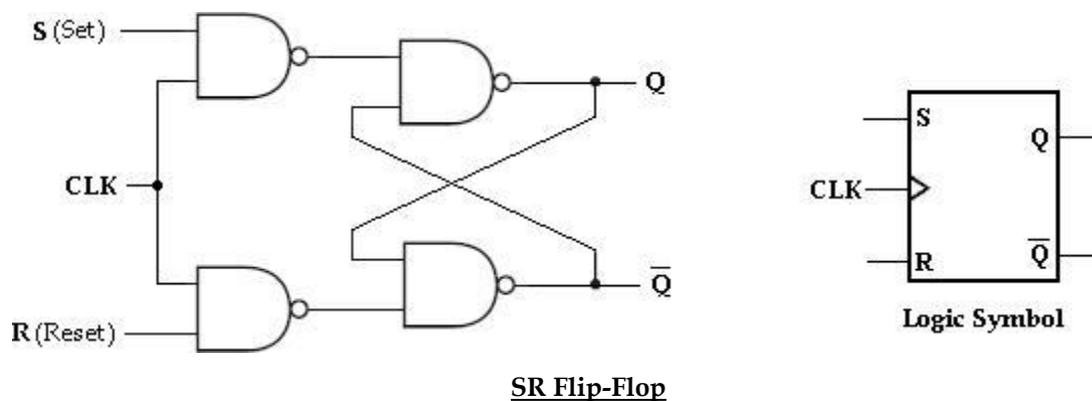
- x S-R Flip-Flop,
- x J-K Flip-Flop,
- x D Flip-Flop,
- x T Flip-Flop.

Although the S-R Flip-Flop is not available in IC form, it is the basis for the D and J-K Flip-Flops. Each type can be either positive edge-triggered (no bubble at C

input) or negative edge-triggered (bubble at C input). The key to identifying an edge-triggered Flip-Flop by its logic symbol is the small triangle inside the block at the clock (C) input. This triangle is called the **dynamic input indicator**.

3.5.1 S-R Flip-Flop

The S and R inputs of the S-R Flip-Flop are called *synchronous* inputs because data on these inputs are transferred to the Flip-Flop's output only on the triggering edge of the clock pulse. The circuit is similar to SR latch except enable signal is replaced by clock pulse (CLK). On the positive edge of the clock pulse, the circuit responds to the S and R inputs.

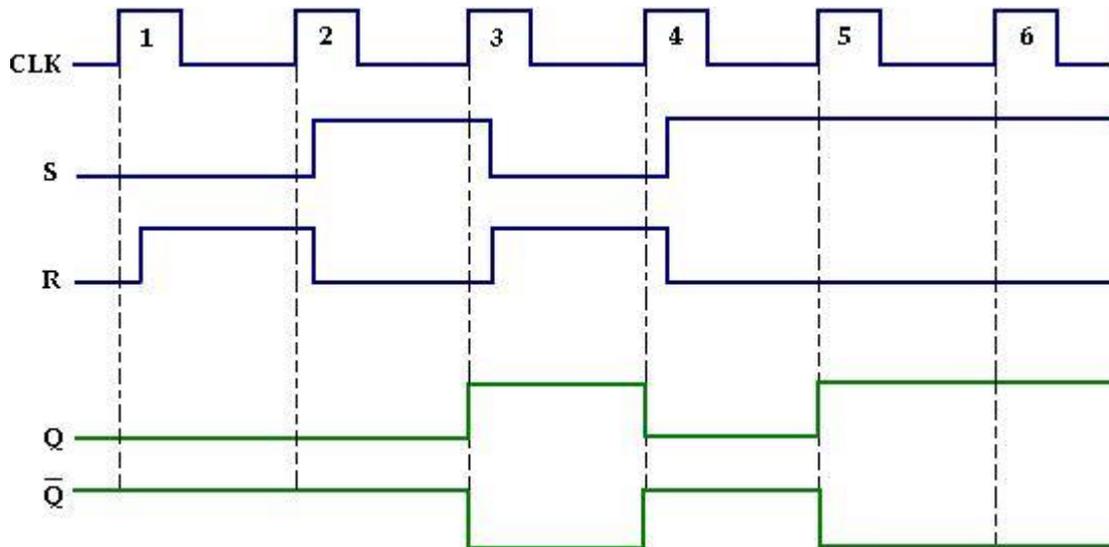


When S is HIGH and R is LOW, the Q output goes HIGH on the triggering edge of the clock pulse, and the Flip-Flop is SET. When S is LOW and R is HIGH, the Q output goes LOW on the triggering edge of the clock pulse, and the Flip-Flop is RESET. When both S and R are LOW, the output does not change from its prior state. An invalid condition exists when both S and R are HIGH.

CLK	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	

1	1	1	0	x	Indeterminate *
1	1	1	1	x	
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

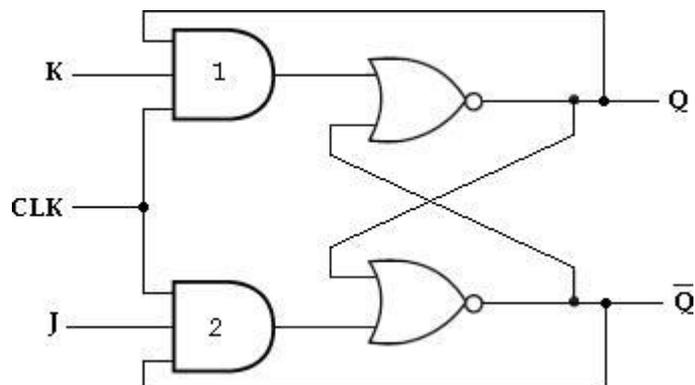
Truth table for SR Flip-Flop



Input and output waveforms of SR Flip-Flop

3.5.2 J-K Flip-Flop:

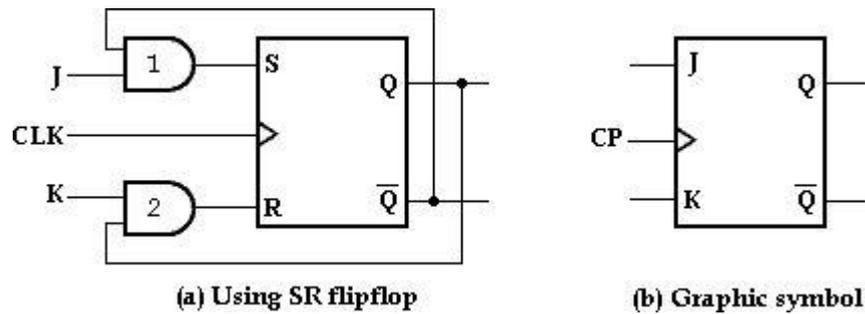
JK means Jack Kilby, Texas Instrument (TI) Engineer, who invented IC in 1958. JK Flip-Flop has two inputs J(set) and K(reset). A JK Flip-Flop can be obtained from the clocked SR Flip-Flop by augmenting two AND gates as shown below.



JK Flip Flop

The data input J and the output Q' are applied to the first AND gate and its output (JQ') is applied to the S input of SR Flip-Flop. Similarly, the data input K and

the output Q are applied to the second AND gate and its output (KQ) is applied to the R input of SR Flip-Flop.



J= K= 0

When J=K= 0, both AND gates are disabled. Therefore clock pulse have no effect, hence the Flip-Flop output is same as the previous output.

J= 0, K= 1

When J= 0 and K= 1, AND gate 1 is disabled i.e., S= 0 and R= 1. This condition will reset the Flip-Flop to 0.

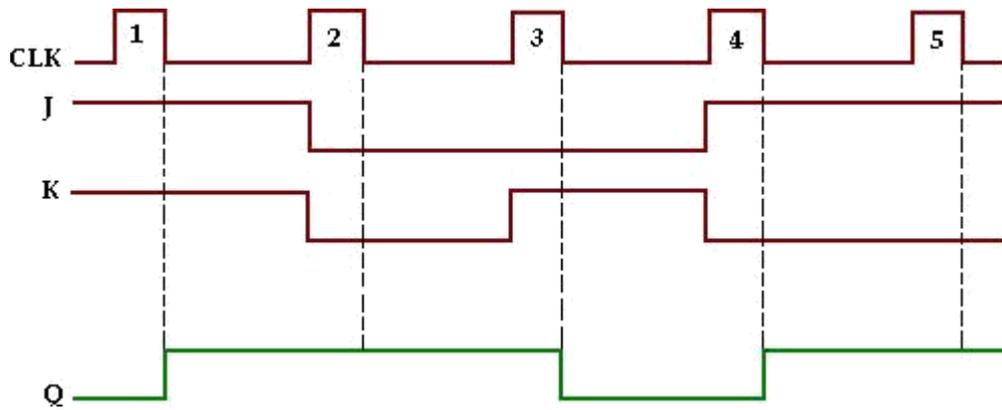
J= 1, K= 0

When J= 1 and K= 0, AND gate 2 is disabled i.e., S= 1 and R= 0. Therefore the Flip-Flop will set on the application of a clock pulse.

J= K= 0

When J=K= 1, it is possible to set or reset the Flip-Flop. If Q is High, AND gate 2 passes on a reset pulse to the next clock. When Q is low, AND gate 1 passes on a set pulse to the next clock. Eitherway, Q changes to the complement of the last state i.e., toggle. Toggle means to switch to the opposite state. The truth table of JK Flip-Flop is given below.

CLK	Inputs		Output	State
	J	K	Q_{n+1}	
1	0	0	Q_n	No Change
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	Q_n'	Toggle



Input and output waveforms of JK Flip-Flop

Characteristic table and Characteristic equation:

The characteristic table for JK Flip-Flop is shown in the table below. From the table, K-map for the next state transition (Q_{n+1}) can be drawn and the simplified logic expression which represents the characteristic equation of JK Flip-Flop can be found.

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Characteristic table

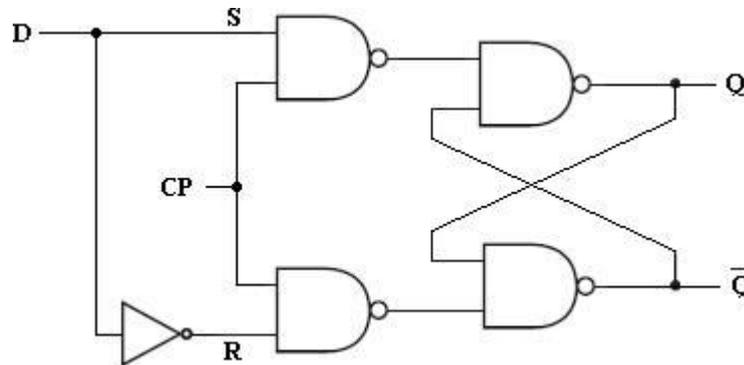
K-map Simplification:

		JK			
Q_n		00	01	11	10
0		0	0	1	1
1		1	0	0	1

Characteristic equation: $Q_{n+1} = JQ' + K'Q$.

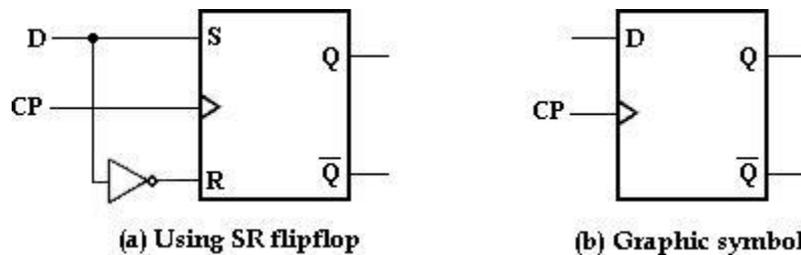
3.5.3 D Flip-Flop:

Like in D latch, in D Flip-Flop the basic SR Flip-Flop is used with complemented inputs. The D Flip-Flop is similar to D-latch except clock pulse is used instead of enable input.



D Flip-Flop

To eliminate the undesirable condition of the indeterminate state in the RS Flip-Flop is to ensure that inputs S and R are never equal to 1 at the same time. This is done by D Flip-Flop. The D (*delay*) Flip-Flop has one input called delay input and clock pulse input. The D Flip-Flop using SR Flip-Flop is shown below.



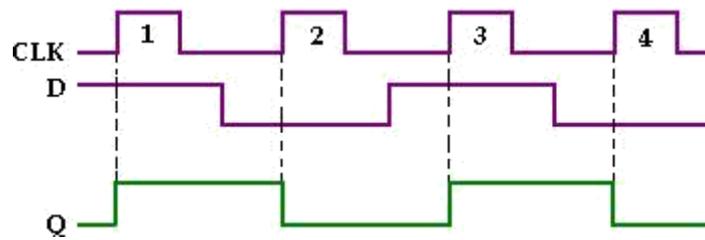
(a) Using SR flipflop

(b) Graphic symbol

The truth table of D Flip-Flop is given below.

Clock	D	Q_{n+1}	State
1	0	0	Reset
1	1	1	Set
0	x	Q_n	No Change

Truth table for D Flip-Flop



Input and output waveforms of clocked D Flip-Flop

Looking at the truth table for D Flip-Flop we can realize that Q_{n+1} function follows the D input at the positive going edges of the clock pulses.

Characteristic table and Characteristic equation:

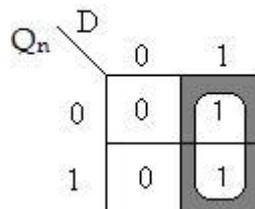
The characteristic table for D Flip-Flop shows that the next state of the Flip-Flop is independent of the present state since Q_{n+1} is equal to D. This means that an input pulse will transfer the value of input D into the output of the Flip-Flop independent of the value of the output before the pulse was applied.

The characteristic equation is derived from K-map.

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic table

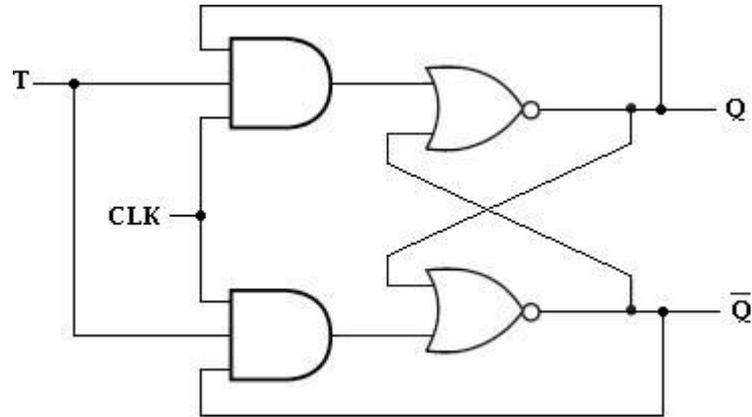
K-map simplification



Characteristic equation: $Q_{n+1} = D$.

3.5.4 T Flip-Flop

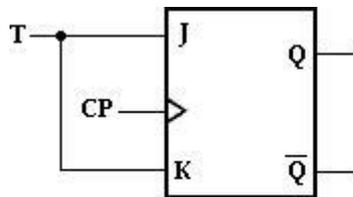
The T (*Toggle*) Flip-Flop is a modification of the JK Flip-Flop. It is obtained from JK Flip-Flop by connecting both inputs J and K together, i.e., single input. Regardless of the present state, the Flip-Flop complements its output when the clock pulse occurs while input T= 1.



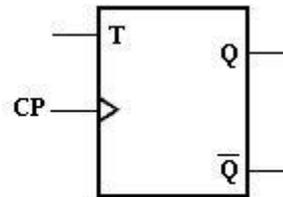
T Flip-Flop

When $T = 0$, $Q_{n+1} = Q_n$, i.e., the next state is the same as the present state and no change occurs.

When $T = 1$, $Q_{n+1} = Q_n'$, i.e., the next state is the complement of the present state.



(a) Using JK flipflop



(b) Graphic symbol

The truth table of T Flip-Flop is given below.

T	Q_{n+1}	State
0	Q_n	No Change
1	Q_n'	Toggle

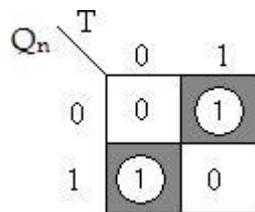
Truth table for T Flip-Flop

Characteristic table and Characteristic equation:

The characteristic table for T Flip-Flop is shown below and characteristic equation is derived using K-map.

Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

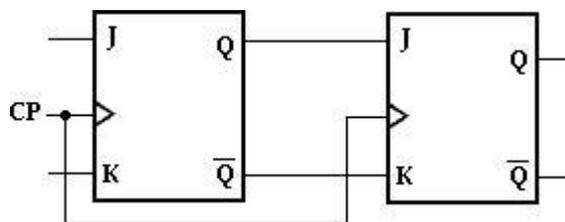
K-map Simplification:



Characteristic equation: $Q_{n+1} = TQ_n' + T'Q_n$.

3.5.5 Master-Slave JK Flip-Flop

A master-slave Flip-Flop is constructed using two separate JK Flip-Flops. The first Flip-Flop is called the master. It is driven by the positive edge of the clock pulse. The second Flip-Flop is called the slave. It is driven by the negative edge of the clock pulse. The logic diagram of a master-slave JK Flip-Flop is shown below.

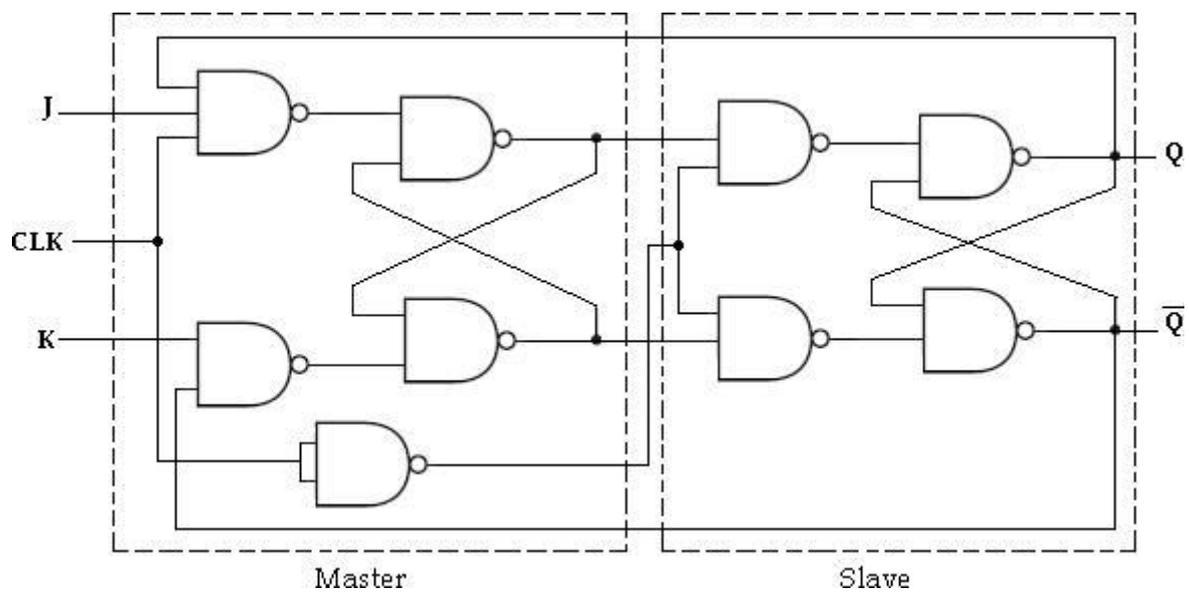


Logic diagram

When the clock pulse has a positive edge, the master acts according to its J-K inputs, but the slave does not respond, since it requires a negative edge at the clock input.

When the clock input has a negative edge, the slave Flip-Flop copies the master outputs. But the master does not respond since it requires a positive edge at its clock input.

The clocked master-slave J-K Flip-Flop using NAND gates is shown below.



Master-Slave JK Flip-Flop

3.6 APPLICATION TABLE (OR) EXCITATION TABLE:

The *characteristic table* is useful for **analysis** and for defining the operation of the Flip-Flop. It specifies the next state (Q_{n+1}) when the inputs and present state are known.

The *excitation or application table* is useful for **design** process. It is used to find the Flip-Flop input conditions that will cause the required transition, when the present state (Q_n) and the next state (Q_{n+1}) are known.

3.6.1 SR Flip-Flop:

Present State	Inputs		Next State
	S	R	
Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	x
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	x

Characteristic Table

Present State	Next State	Inputs		Inputs	
		S	R	S	R
Q_n	Q_{n+1}	S	R	S	R
0	0	0	0	0	x
0	0	0	1		
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	0	x	0
1	1	1	0		

Modified Table

Present State	Next State	Inputs	
		S	R
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Excitation Table

The above table presents the excitation table for SR Flip-Flop. It consists of present state (Q_n), next state (Q_{n+1}) and a column for each input to show how the required transition is achieved.

There are 4 possible transitions from present state to next state. The required Input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol 'x' denotes the don't care condition, it does not matter whether the input is 0 or 1.

3.6.2 JK Flip-Flop:

Present State	Inputs		Next State
	J	K	
Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Characteristic Table

Present State	Next State	Inputs		Inputs	
		J	K	J	K
Q_n	Q_{n+1}	J	K	J	K
0	0	0	0	0	x
0	0	0	1		
0	1	1	0	1	x
0	1	1	1		
1	0	0	1	x	1
1	0	1	1		
1	1	0	0	x	0
1	1	1	0		

Modified Table

Present State	Next State	Inputs	
		J	K
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation Table

3.6.3 D Flip-Flop

Present State	Input	Next State
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic Table

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation Table

3.6.4 T Flip-Flop

Present State	Input	Next State
Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic Table

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Modified Table

3.7 REALIZATION OF ONE FLIP-FLOP USING OTHER FLIP-FLOPS

It is possible to convert one Flip-Flop into another Flip-Flop with some additional gates or simply doing some extra connection. The realization of one Flip-Flop using other Flip-Flops is implemented by the use of characteristic tables and excitation tables. Let us see few conversions among Flip-Flops.

- ✦ SR Flip-Flop to D Flip-Flop
- ✦ SR Flip-Flop to JK Flip-Flop
- ✦ SR Flip-Flop to T Flip-Flop
- ✦ JK Flip-Flop to T Flip-Flop
- ✦ JK Flip-Flop to D Flip-Flop
- ✦ D Flip-Flop to T Flip-Flop
- ✦ T Flip-Flop to D Flip-Flop

3.7.1 SR Flip-Flop to D Flip-Flop:

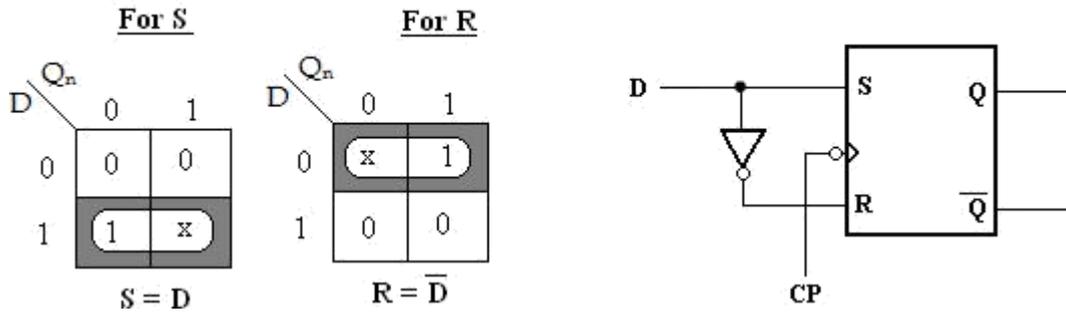
- x Write the characteristic table for required Flip-Flop (D Flip-Flop).
- x Write the excitation table for given Flip-Flop (SR Flip-Flop).
- x Determine the expression for the given Flip-Flop inputs (S and R) by using K- map.
- x Draw the Flip-Flop conversion logic diagram to obtain the required Flip-Flop (D Flip-Flop) by using the above obtained expression.

The excitation table for the above conversion is

Required Flip-Flop (D)			Given Flip-Flop (SR)	
Input	Present state	Next state	Flip-Flop Inputs	
D	Q_n	Q_{n+1}	S	R
0	0	0	0	x
0	1	0	0	1
1	0	1	1	0
1	1	1	x	0

K-map simplification

Logic diagram



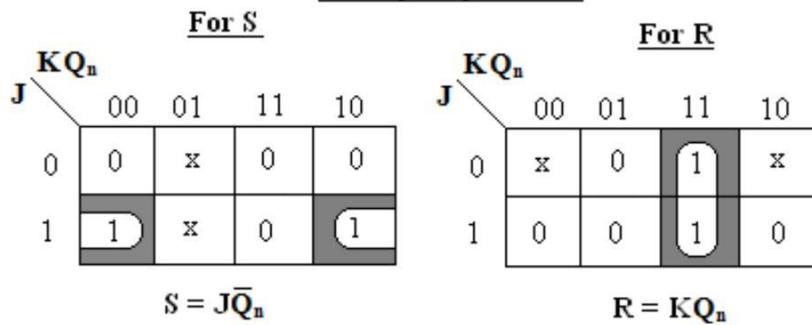
D Flip-Flop

3.7.2 SR Flip-Flop to JK Flip-Flop

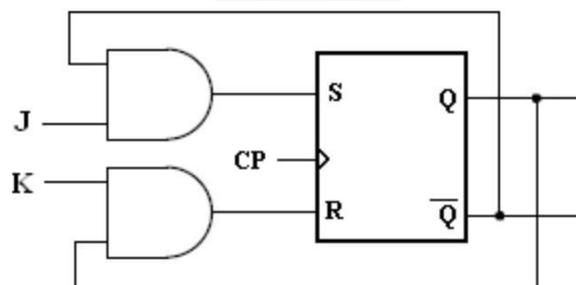
The excitation table for the above conversion is,

Inputs		Present state	Next state	Flip-Flop Input	
J	K	Q_n	Q_{n+1}	S	R
0	0	0	0	0	x
0	0	1	1	x	0
0	1	0	0	0	x
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	x	0
1	1	0	1	1	0
1	1	1	0	0	1

K-map simplification



Logic diagram



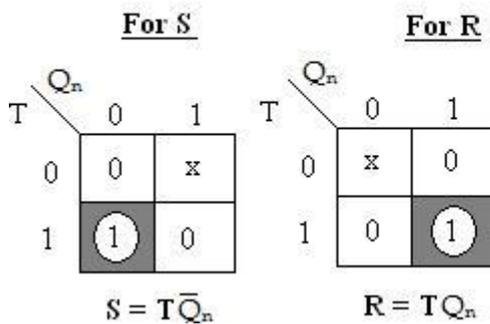
JK Flip-Flop

2.7.3 SR Flip-Flop to T Flip-Flop

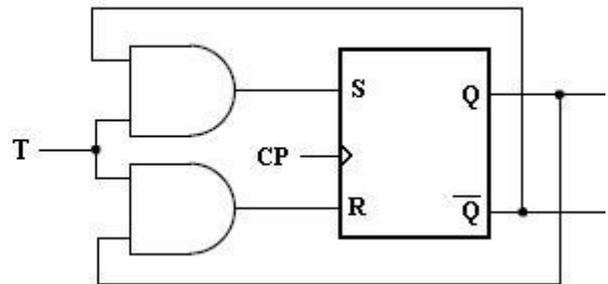
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
			S	R
0	0	0	0	x
0	1	1	x	0
1	0	1	1	0
1	1	0	0	1

K-map simplification



Logic diagram

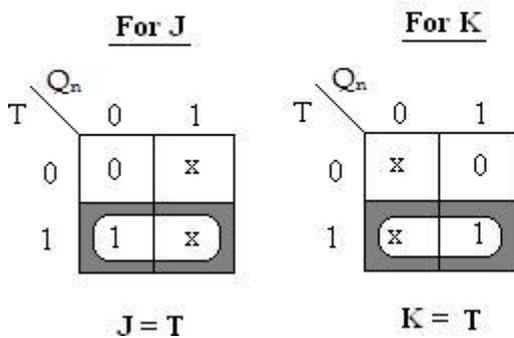


3.7.4 JK Flip-Flop to T Flip-Flop

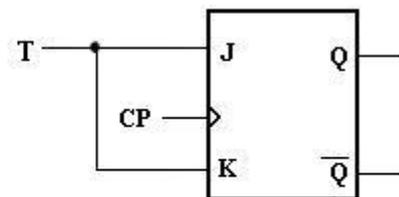
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
			J	K
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

K-map simplification



Logic diagram



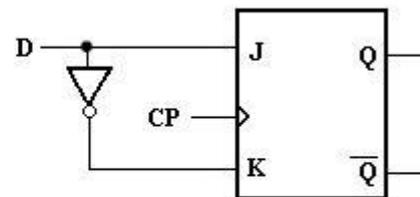
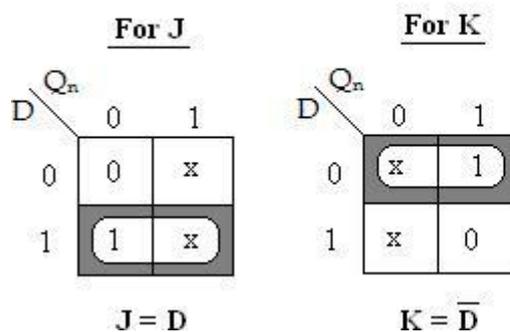
JK Flip-Flop to D Flip-Flop

The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
D	Q_n	Q_{n+1}	J	K
0	0	0	0	x
0	1	0	x	1
1	0	1	1	x
1	1	1	x	0

K-map simplification

Logic diagram



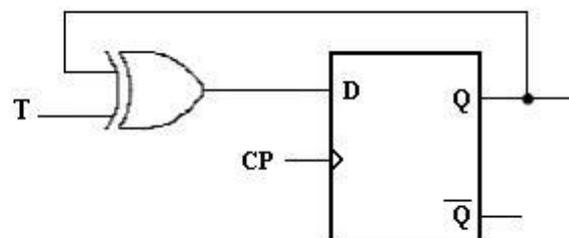
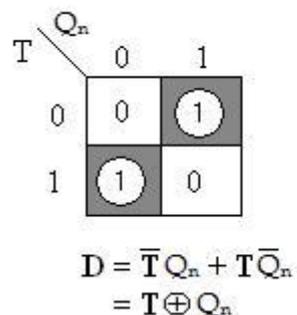
D Flip-Flop to T Flip-Flop

The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Input
T	Q_n	Q_{n+1}	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

K-map simplification

Logic diagram

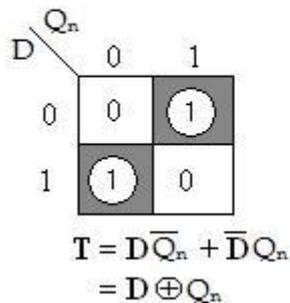


T Flip-Flop to D Flip-Flop

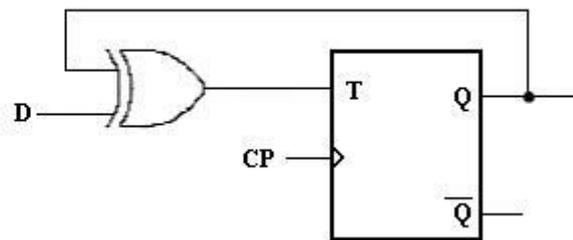
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Input
D	Q_n	Q_{n+1}	T
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

K-map simplification



Logic diagram



3.8 CLASSIFICATION OF SYNCHRONOUS SEQUENTIAL CIRCUIT:

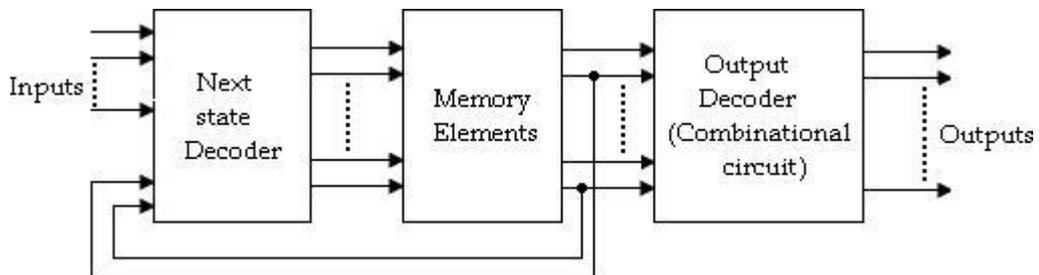
In synchronous or clocked sequential circuits, clocked Flip-Flops are used as memory elements, which change their individual states in synchronism with the periodic clock signal. Therefore, the change in states of Flip-Flop and change in state of the entire circuits occur at the transition of the clock signal.

The synchronous or clocked sequential networks are represented by two models.

- x **Moore model:** The output depends only on the present state of the Flip-Flops.
- x **Mealy model:** The output depends on both the present state of the Flip-Flops and on the inputs.

3.8.1 Moore model:

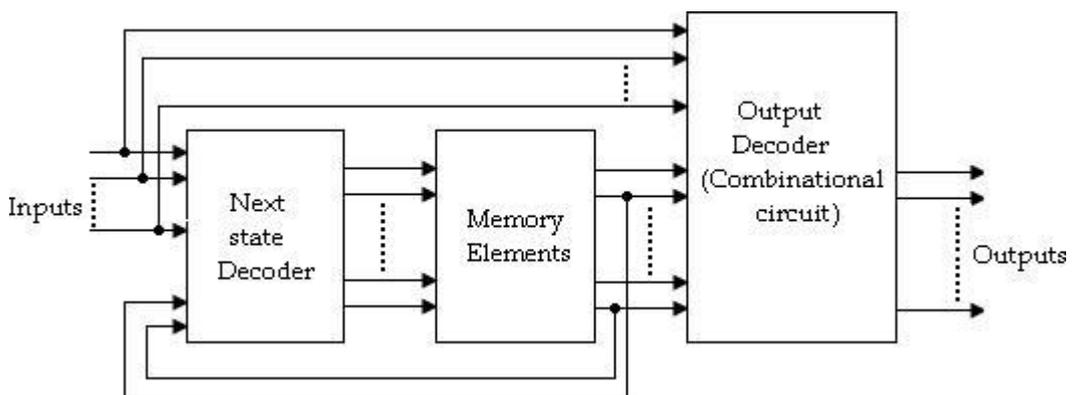
In the Moore model, the outputs are a function of the present state of the Flip-Flops only. The output depends only on present state of Flip-Flops, it appears only after the clock pulse is applied, i.e., it varies in synchronism with the clock input.



Moore model

3.8.2 Mealy model:

In the Mealy model, the outputs are functions of both the present state of the Flip-Flops and inputs.



Mealy model

3.8.3 Difference between Moore and Mealy model

Sl.No	Moore model	Mealy model
1	Its output is a function of present state only.	Its output is a function of present state as well as present input.
2	Input changes does not affect the output.	Input changes may affect the output of the circuit.
3	It requires more number of states for implementing same function.	It requires less number of states for implementing same function.

3.9 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUIT:

The behavior of a sequential circuit is determined from the inputs, outputs and the state of its Flip-Flops. The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or diagram from the time sequence of inputs, outputs and internal states.

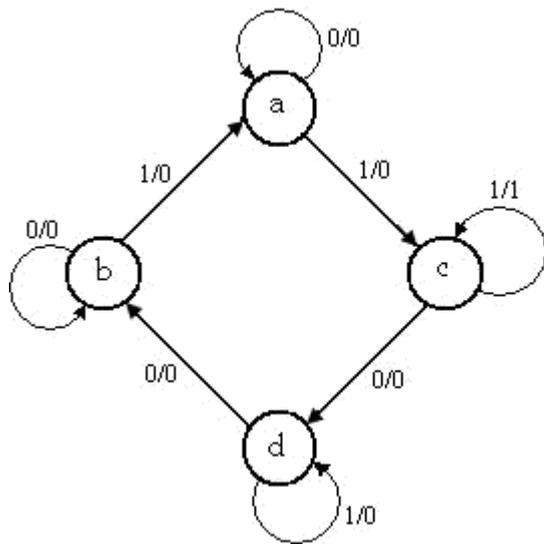
Before going to see the analysis and design examples, we first understand the state diagram, state table.

3.9.1 State Diagram

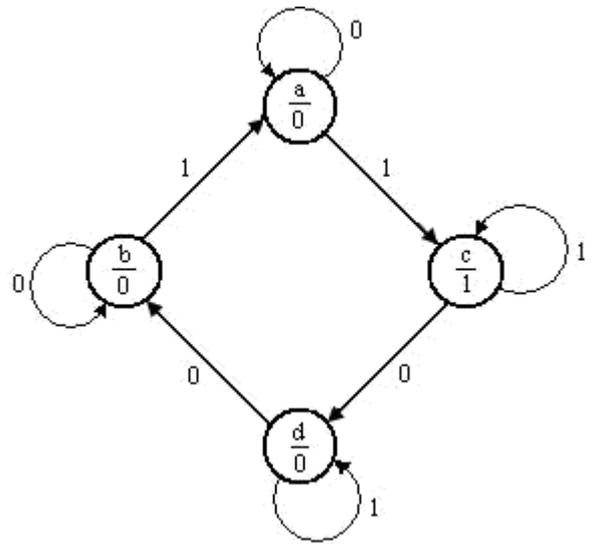
State diagram is a pictorial representation of a behavior of a sequential circuit.

- ▶ In the state diagram, a state is represented by a circle and the transition between states is indicated by directed lines connecting the circles.
- ▶ A directed line connecting a circle with circle with itself indicates that next state is same as present state.
- ▶ The binary number inside each circle identifies the state represented by the circle.
- ▶ The directed lines are labeled with two binary numbers separated by a symbol '/'. The input value that causes the state transition is labeled first and the output value during the present state is labeled after the symbol '/'.

In case of Moore circuit, the directed lines are labeled with only one binary number representing the state of the input that causes the state transition. The output state is indicated within the circle, below the present state because output state depends only on present state and not on the input.



State diagram for Mealy circuit



State diagram for Moore circuit

3.9.2 State Table

State table represents relationship between input, output and Flip-Flop states.

- ▮ It consists of three sections labeled present state, next state and output.
 - The present state designates the state of Flip-Flops before the occurrence of a clock pulse, and the output section gives the values of the output variables during the present state.
 - Both the next state and output sections have two columns representing two possible input conditions: $X=0$ and $X=1$.

Present state	Next state		Output	
	X=0	X=1	X=0	X=1
AB	AB	AB	Y	Y
a	a	c	0	0
b	b	a	0	0
c	d	c	0	1
d	b	d	0	0

- ▮ In case of Moore circuit, the output section has only one column since output does not depend on input.

Present state	Next state		Output Y
	X= 0	X= 1	
AB	AB	AB	
a	a	c	0
b	b	a	0
c	d	c	1
d	b	d	0

2.9.3 State Equation

It is an algebraic expression that specifies the condition for a Flip-Flop state transition.

The Flip-Flops may be of any type and the logic diagram may or may not include combinational circuit gates.

3.9.4 ANALYSIS PROCEDURE

The synchronous sequential circuit analysis is summarized as given below:

1. Assign a state variable to each Flip-Flop in the synchronous sequential circuit.
2. Write the excitation input functions for each Flip-Flop and also write the Moore/ Mealy output equations.
3. Substitute the excitation input functions into the bistable equations for the Flip-Flops to obtain the next state output equations.
4. Obtain the state table and reduced form of the state table.
5. Draw the state diagram by using the second form of the state table.

3.9.5 Analysis of Mealy Model

1. A sequential circuit has two JK Flip-Flops A and B, one input (x) and one output (y). the Flip-Flop input functions are,

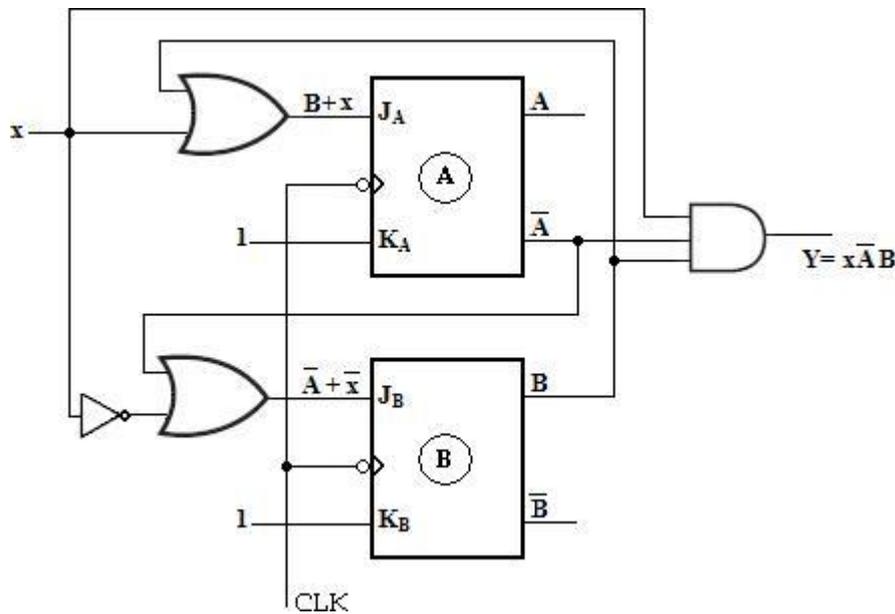
$$J_A = B + x \qquad J_B = A' + x'$$

$$K_A = 1 \qquad K_B = 1$$

and the circuit output function, $Y = xA'B$.

- Draw the logic diagram of the Mealy circuit,
- Tabulate the state table,
- Draw the state diagram.

Soln:



State table:

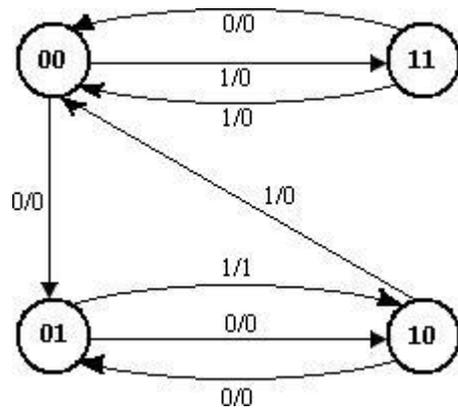
To obtain the next-state values of a sequential circuit with JK Flip-Flops, use the JK Flip-Flop characteristics table.

Present state		Input x	Flip-Flop Inputs				Next state		Output Y = xA'B
A	B		$J_A = B + x$	$K_A = 1$	$J_B = A' + x'$	$K_B = 1$	A(t+1)	B(t+1)	
0	0	0	0	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	0
0	1	0	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	1	1	0	1	0
1	0	1	1	1	0	1	0	0	0
1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	1	0	0	0

Present state		Next state				Output	
		x= 0		x= 1		x= 0	x= 1
A	B	A	B	A	B	y	y
0	0	0	1	1	1	0	0
0	1	1	0	1	0	0	1
1	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0

Second form of state table

State Diagram:



State Diagram

2. A sequential circuit with two 'D' Flip-Flops A and B, one input (x) and one output (y). the Flip-Flop input functions are:

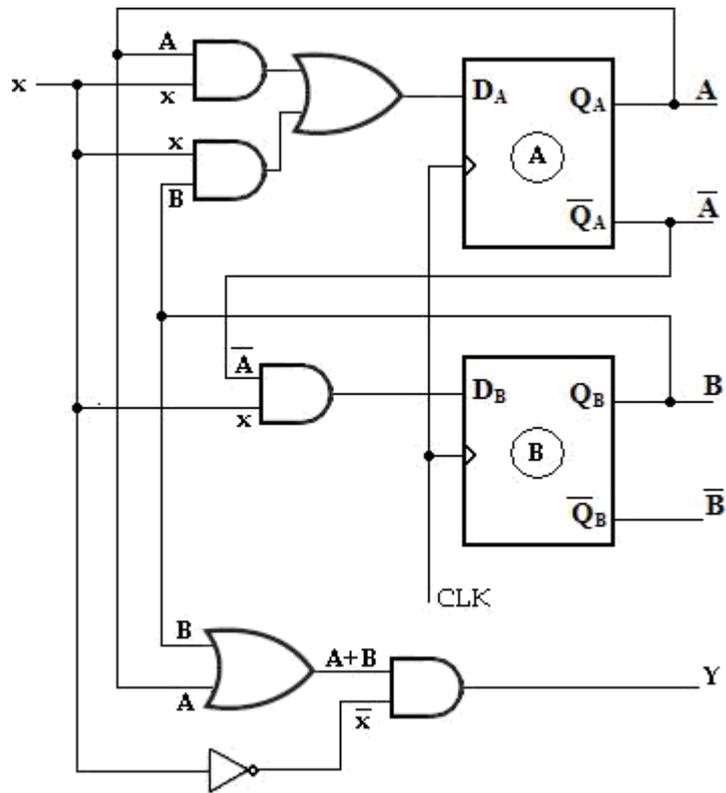
$$D_A = Ax + Bx$$

$$D_B = A'x \text{ and the circuit output function is,}$$

$$Y = (A + B) x'$$

- (a) Draw the logic diagram of the circuit,
- (b) Tabulate the state table,
- (c) Draw the state diagram.

Soln:



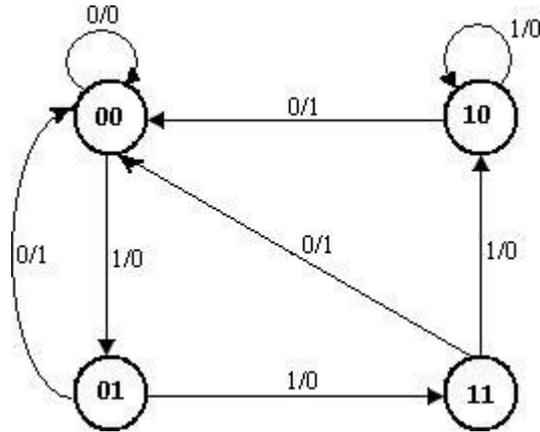
State Table:

Present state		Input	Flip-Flop Inputs		Next state		Output
A	B	x	$D_A = Ax + Bx$	$D_B = A'x$	A(t+1)	B(t+1)	$Y = (A+B)x'$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	1
1	1	1	1	0	1	0	0

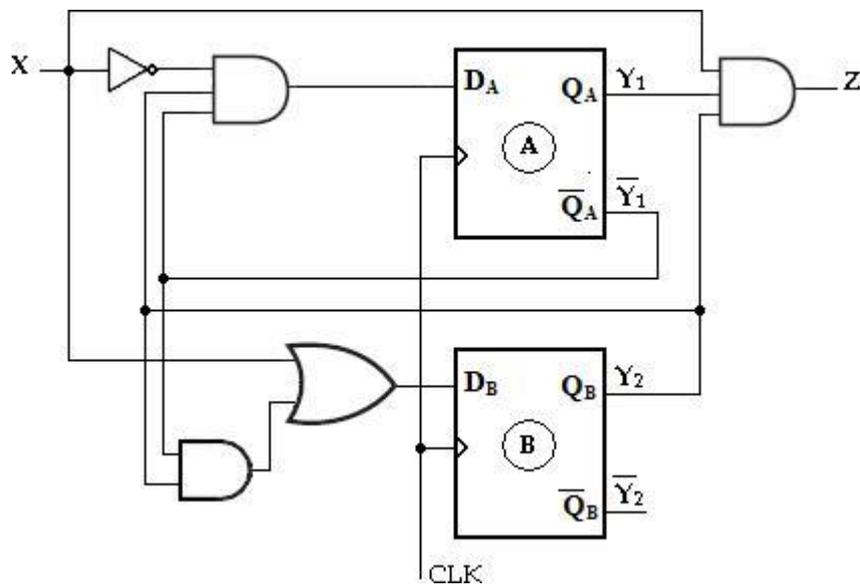
Present state		Next state				Output	
		x=0		x=1		x=0	x=1
A	B	A	B	A	B	Y	Y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

Second form of state table

State Diagram:



3. Analyze the synchronous Mealy machine and obtain its state diagram.



Soln:

The given synchronous Mealy machine consists of two D Flip-Flops, one inputs and one output.

The Flip-Flop input functions are,

$$D_A = Y_1'Y_2X'$$

$$D_B = X + Y_1Y_2$$

The circuit output function is, $Z = Y_1Y_2X$

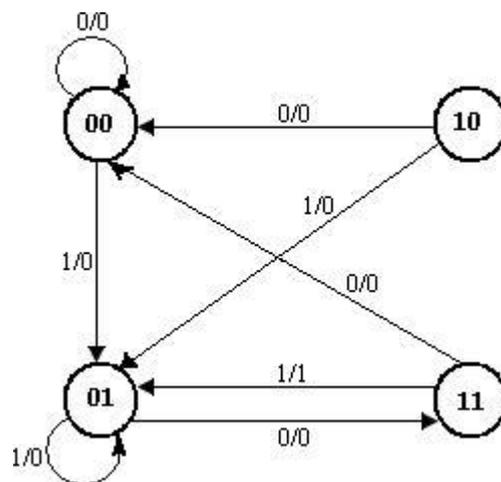
State Table:

Present state		Input	Flip-Flop Inputs		Next state		Output
Y ₁	Y ₂	X	D _A = Y ₁ 'Y ₂ X'	D _B = X + Y ₁ 'Y ₂	Y ₁ (t+1)	Y ₂ (t+1)	Z = Y ₁ Y ₂ X
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Present state		Next state				Output	
		X = 0		X = 1		X = 0	X = 1
Y ₁	Y ₂	Y ₁	Y ₂	Y ₁	Y ₂	Z	Z
0	0	0	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	0
1	1	0	0	0	1	0	1

Second form of state table

State Diagram:



4. A sequential circuit has two JK Flop-Flops A and B, two inputs x and y and one output z. The Flip-Flop input equation and circuit output equations are

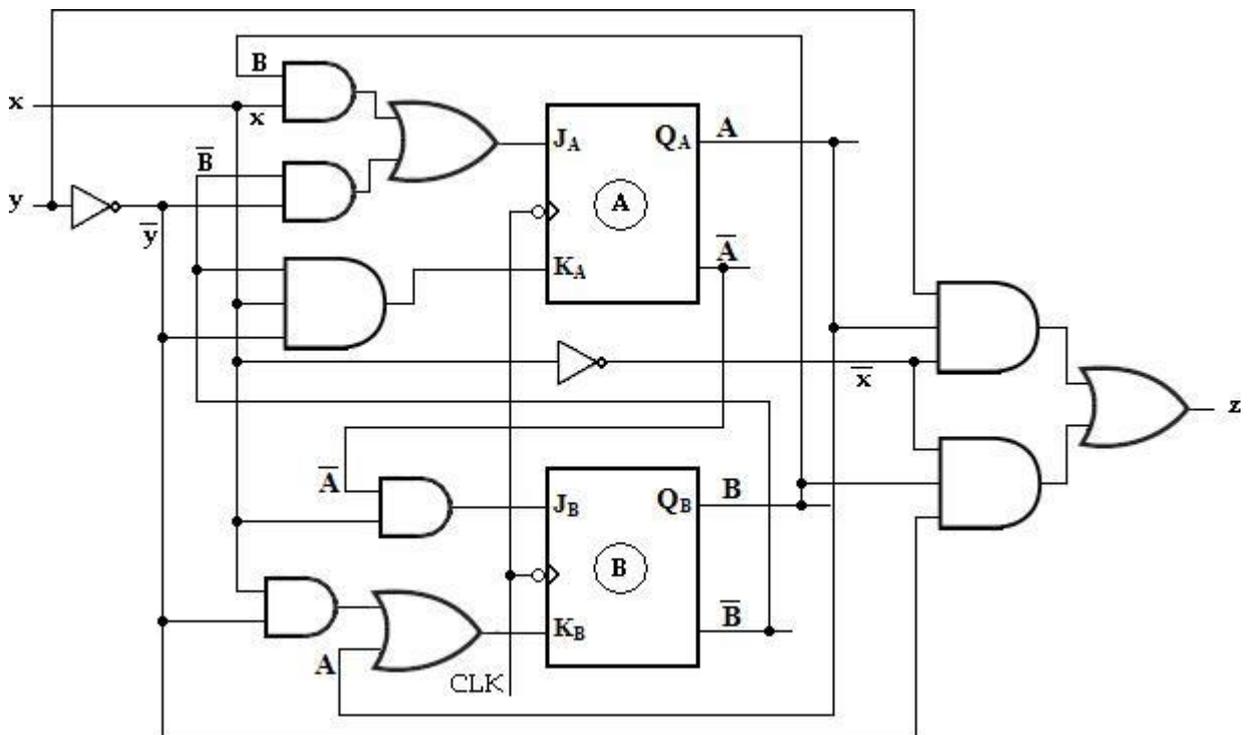
$$J_A = Bx + B'y' \quad K_A = B'xy'$$

$$J_B = A'x \quad K_B = A + xy'$$

$$z = Ax'y' + Bx'y'$$

- Draw the logic diagram of the circuit
- Tabulate the state table.
- Derive the state equation.

State diagram:



State table:

To obtain the next-state values of a sequential circuit with JK Flip-Flop, use the JK Flip-Flop characteristic table,

Present state		Input		Flip-Flop Inputs				Next state		Output
A	B	x	y	$J_A = Bx + B'y'$	$K_A = B'xy'$	$J_B = A'x$	$K_B = A + xy'$	A(t+1)	B(t+1)	z
0	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1	1	1	0
0	0	1	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	0	0	1
0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	1	0	1	1	1	1	0
0	1	1	1	1	0	1	0	1	1	0
1	0	0	0	1	0	0	1	1	0	1
1	0	0	1	0	0	0	1	1	0	0
1	0	1	0	1	1	0	1	0	0	0
1	0	1	1	0	0	0	1	1	0	0
1	1	0	0	0	0	0	1	1	0	1
1	1	0	1	0	0	0	1	1	0	0
1	1	1	0	1	0	0	1	1	0	0
1	1	1	1	1	0	0	1	1	0	0

State Equation:

For A(t+1)

AB \ xy	00	01	11	10
00	1	0	0	1
01	0	0	1	1
11	1	1	1	1
10	1	1	1	0

$$A(t+1) = Ax' + Ay + Bx + A'B'y'$$

For B(t+1)

AB \ xy	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	0	0

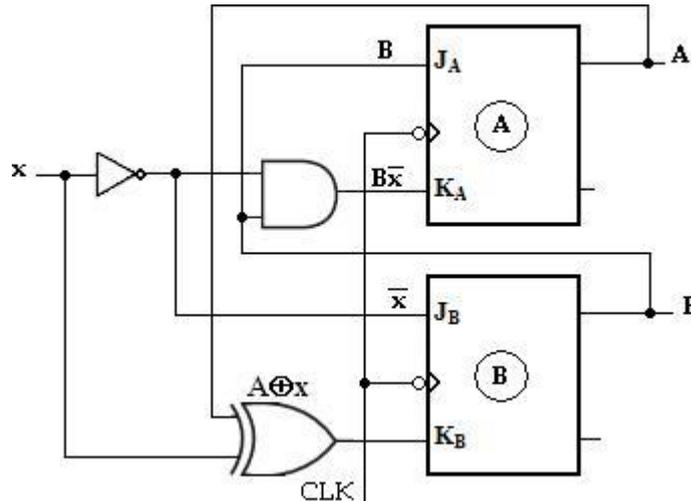
$$B(t+1) = A'x$$

5. A sequential circuit has two JK Flip-Flop A and B. the Flip-Flop input functions

$$\begin{aligned} \text{are: } J_A &= B & J_B &= x' \\ K_A &= Bx' & K_B &= A \oplus x. \end{aligned}$$

- (a) Draw the logic diagram of the circuit,
- (b) Tabulate the state table,
- (c) Draw the state diagram.

Logic diagram:



The output function is not given in the problem. The output of the Flip-Flops may be considered as the output of the circuit.

State table:

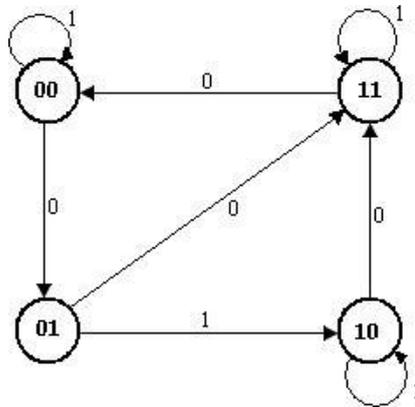
To obtain the next-state values of a sequential circuit with JK Flip-Flop, use the JK Flip-Flop characteristic table.

Present state		Input	Flip-Flop Inputs				Next state	
A	B	x	$J_A = B$	$K_A = Bx'$	$J_B = x'$	$K_B = A \oplus x$	$A(t+1)$	$B(t+1)$
0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	0
0	1	0	1	1	1	0	1	1
0	1	1	1	0	0	1	1	0
1	0	0	0	0	1	1	1	1
1	0	1	0	0	0	0	1	0
1	1	0	1	1	1	1	0	0
1	1	1	1	0	0	0	1	1

Present state		Next state			
		X=0		X=1	
A	B	A	B	A	B
0	0	0	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	0	0	1	1

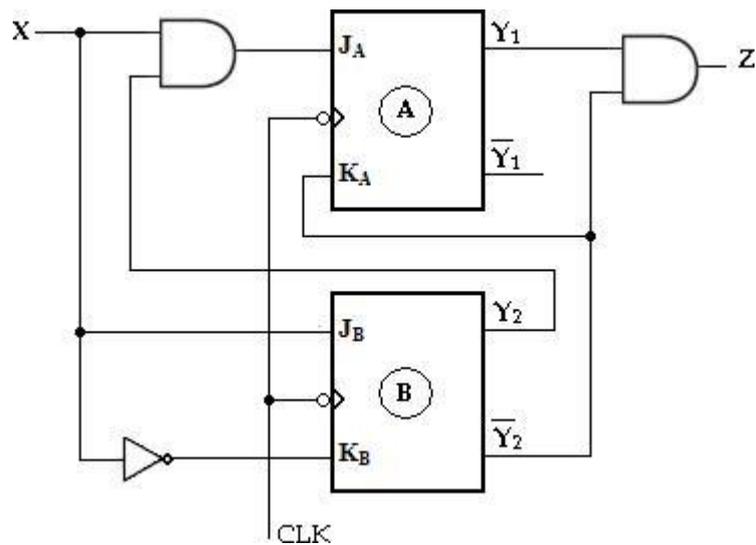
Second form of state table

State Diagram:



3.9.6 Analysis of Moore Model

6. Analyze the synchronous Moore circuit and obtain its state diagram.



Soln:

Using the assigned variable Y_1 and Y_2 for the two JK Flip-Flops, we can write the four excitation input equations and the Moore output equation as follows:

$$J_A = Y_2 X \quad ; \quad K_A = Y_2'$$

$$J_B = X \quad ; \quad K_B = X' \quad \text{and output function, } Z = Y_1 Y_2'$$

State table:

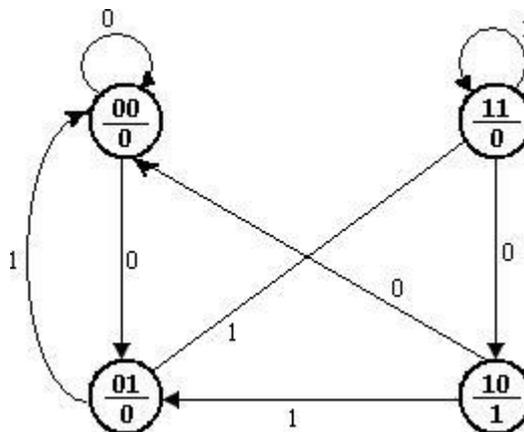
Present state		Input	Flip-Flop Inputs				Next state		Output
Y ₁	Y ₂	X	J _A = Y ₂ X	K _A = Y ₂ '	J _B = X	K _B = X'	Y ₁ (t+1)	Y ₂ (t+1)	Z = Y ₁ Y ₂ '
0	0	0	0	1	0	1	0	0	0
0	0	1	0	1	1	0	0	1	0
0	1	0	0	0	0	1	0	0	0
0	1	1	1	0	1	0	1	1	0
1	0	0	0	1	0	1	0	0	1
1	0	1	0	1	1	0	0	1	1
1	1	0	0	0	0	1	1	0	0
1	1	1	1	0	1	0	1	1	0

Present state		Next state				Output Y
		X = 0		X = 1		
Y ₁	Y ₂	Y ₁	Y ₂	Y ₁	Y ₂	
0	0	0	0	0	1	0
0	1	0	0	1	1	0
1	0	0	0	0	1	1
1	1	1	0	1	1	0

Second form of state table

State Diagram:

Here the output depends on the present state only and is independent of the input. The two values inside each circle separated by a slash are for the present state and output.



7. A sequential circuit has two T Flip-Flop A and B. The Flip-Flop input functions are:

$$T_A = Bx$$

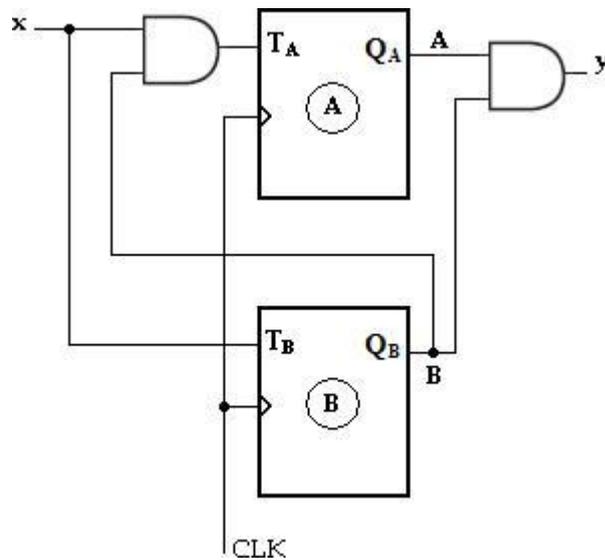
$$T_B = x$$

$$y = AB$$

- Draw the logic diagram of the circuit,
- Tabulate the state table,
- Draw the state diagram.

Soln:

Logic diagram:



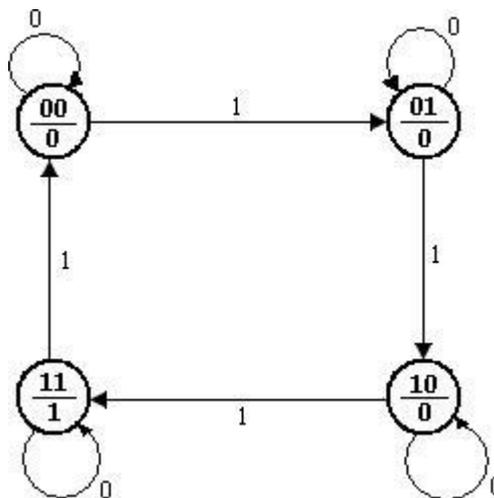
State table

Present state		Input	Flip-Flop Inputs		Next state		Output
A	B	x	$T_A = Bx$	$T_B = x$	A (t+1)	B (t+1)	y = AB
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1	1	1	0	0	1

Present state		Next state				Output	
		x= 0		x= 1		x= 0	x= 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	1	1

Second form of state table

State Diagram:



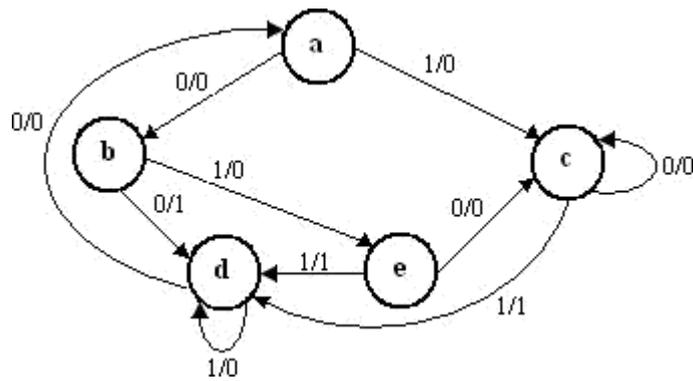
3.10 STATE REDUCTION/ MINIMIZATION

The state reduction is used to avoid the redundant states in the sequential circuits. The reduction in redundant states reduces the number of required Flip-Flops and logic gates, reducing the cost of the final circuit.

The two states are said to be redundant or equivalent, if every possible set of inputs generate exactly same output and same next state. When two states are equivalent, one of them can be removed without altering the input-output relationship.

Since 'n' Flip-Flops produced 2^n state, a reduction in the number of states may result in a reduction in the number of Flip-Flops.

The need for state reduction or state minimization is explained with one example.



State diagram

Step 1: Determine the state table for given state diagram

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	b	c	0	0
b	d	e	1	0
c	c	d	0	1
d	a	d	0	0
e	c	d	0	1

State table

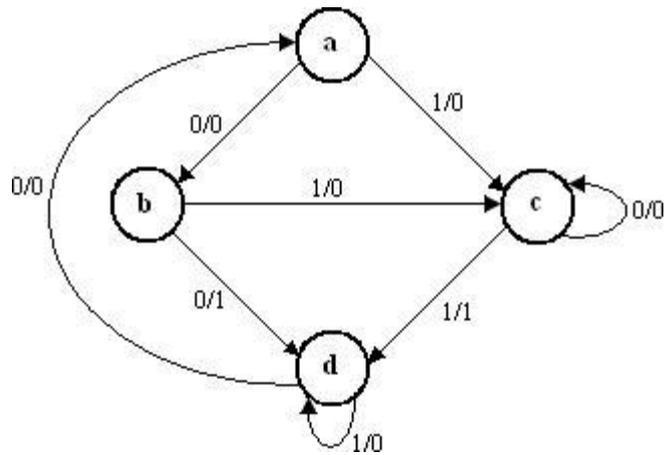
Step 2: Find equivalent states

From the above state table **c** and **e** generate exactly same next state and same output for every possible set of inputs. The state **c** and **e** go to next states **c** and **d** and have outputs 0 and 1 for $x=0$ and $x=1$ respectively. Therefore state **e** can be removed and replaced by **c**. The final reduced state table is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	b	c	0	0
b	d	c	1	0
c	c	d	0	1
d	a	d	0	0

Reduced state table

The state diagram for the reduced table consists of only four states and is shown below.



Reduced state diagram

1. Reduce the number of states in the following state table and tabulate the reduced state table.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Soln:

From the above state table **e** and **g** generate exactly same next state and same output for every possible set of inputs. The state **e** and **g** go to next states **a** and **f** and have outputs 0 and 1 for $x=0$ and $x=1$ respectively. Therefore state **g** can be removed and replaced by **e**.

The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Reduced state table-1

Now states d and f are equivalent. Both states go to the same next state (e, f) and have same output (0, 1). Therefore one state can be removed; f is replaced by d. The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

Reduced state table-2

Thus 7 states are reduced into 5 states.

2. Determine a minimal state table equivalent furnished below

Present state	Next state	
	X= 0	X= 1
1	1, 0	1, 0
2	1, 1	6, 1
3	4, 0	5, 0
4	1, 1	7, 0
5	2, 0	3, 0
6	4, 0	5, 0
7	2, 0	3, 0

Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
1	1	1	0	0
2	1	6	1	1
3	4	5	0	0
4	1	7	1	0
5	2	3	0	0
6	4	5	0	0
7	2	3	0	0

From the above state table, 5 and 7 generate exactly same next state and same output for every possible set of inputs. The state 5 and 7 go to next states 2 and 3 and have outputs 0 and 0 for $x=0$ and $x=1$ respectively. Therefore state 7 can be removed and replaced by 5.

Similarly, 3 and 6 generate exactly same next state and same output for every possible set of inputs. The state 3 and 6 go to next states 4 and 5 and have outputs 0 and 0 for $x=0$ and $x=1$ respectively. Therefore state 6 can be removed and replaced by 3.

The final reduced state table is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
1	1	1	0	0
2	1	3	1	1
3	4	5	0	0
4	1	5	1	0
5	2	3	0	0

Reduced state table

Thus 7 states are reduced into 5 states.

3. Minimize the following state table.

Present state	Next state	
	X= 0	X= 1
A	D, 0	C, 1
B	E, 1	A, 1
C	H, 1	D, 1
D	D, 0	C, 1
E	B, 0	G, 1
F	H, 1	D, 1
G	A, 0	F, 1
H	C, 0	A, 1
I	G, 1	H, 1

Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	D	C	0	1
B	E	A	1	1
C	H	D	1	1
D	D	C	0	1
E	B	G	0	1
F	H	D	1	1
G	A	F	0	1
H	C	A	0	1
I	G	H	1	1

From the above state table, **A** and **D** generate exactly same next state and same output for every possible set of inputs. The state **A** and **D** go to next states **D** and **C** and have outputs 0 and 1 for $x=0$ and $x=1$ respectively. Therefore state **D** can be removed and replaced by **A**. Similarly, **C** and **F** generate exactly same next state and same output for every possible set of inputs. The state **C** and **F** go to next states **H** and **D** and have outputs 1 and 1 for $x=0$ and $x=1$ respectively. Therefore state **F** can be removed and replaced by **C**.

The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	A	C	0	1
B	E	A	1	1
C	H	A	1	1
E	B	G	0	1
G	A	C	0	1
H	C	A	0	1
I	G	H	1	1

Reduced state table-1

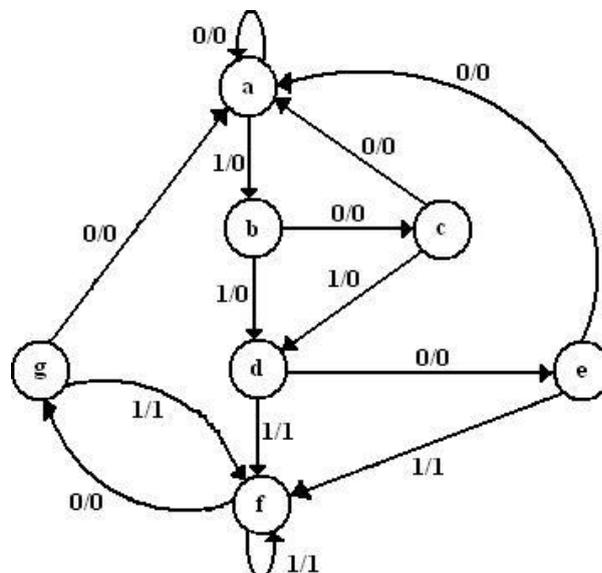
From the above reduced state table-1, A and G generate exactly same next state and same output for every possible set of inputs. The state A and G go to next states A and C and have outputs 0 and 1 for $x=0$ and $x=1$ respectively. Therefore state G can be removed and replaced by A. The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	A	C	0	1
B	E	A	1	1
C	H	A	1	1
E	B	A	0	1
H	C	A	0	1
I	A	H	1	1

Reduced state table-2

Thus 9 states are reduced into 6 states.

4. Reduce the following state diagram.



Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

State table

From the above state table **e** and **g** generate exactly same next state and same output for every possible set of inputs. The state **e** and **g** go to next states **a** and **f** and have outputs 0 and 1 for $x=0$ and $x=1$ respectively. Therefore state **g** can be removed and replaced by **e**. The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Reduced state table-1

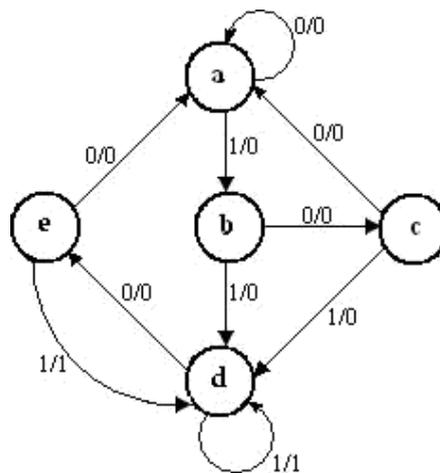
Now states **d** and **f** are equivalent. Both states go to the same next state (**e**, **f**) and have same output (0, 1). Therefore one state can be removed; **f** is replaced by **d**. The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

Reduced state table-2

Thus 7 states are reduced into 5 states.

The state diagram for the reduced state table-2 is,



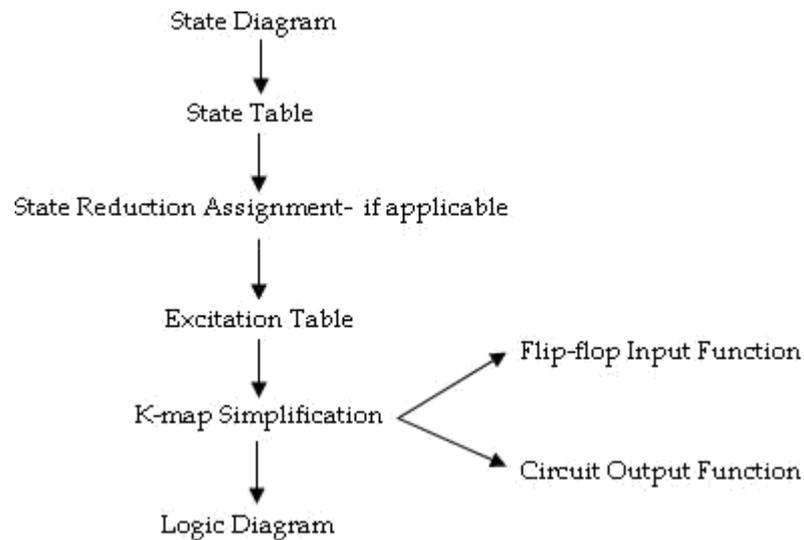
Reduced state diagram

3.11 DESIGN OF SYNCHRONOUS SEQUENTIAL CIRCUITS:

A synchronous sequential circuit is made up of number of Flip-Flops and combinational gates. The design of circuit consists of choosing the Flip-Flops and then finding a combinational gate structure together with the Flip-Flops. The number of Flip-Flops is determined from the number of states needed in the circuit. The combinational circuit is derived from the state table.

3.11.1 Design procedure:

1. The given problem is determined with a state diagram.
2. From the state diagram, obtain the state table.
3. The number of states may be reduced by state reduction methods (if applicable).
4. Assign binary values to each state (Binary Assignment) if the state table contains letter symbols.
5. Determine the number of Flip-Flops and assign a letter symbol (A, B, C,...) to each.
6. Choose the type of Flip-Flop (SR, JK, D, T) to be used.
7. From the state table, circuit excitation and output tables.
8. Using K-map or any other simplification method, derive the circuit output functions and the Flip-Flop input functions.
9. Draw the logic diagram.



The type of Flip-Flop to be used may be included in the design specifications or may depend what is available to the designer. Many digital systems are constructed with JK Flip-Flops because they are the most versatile available. The selection of inputs is given as follows.

Flip-Flop	Application
JK	General Applications
D	Applications requiring transfer of data
T	(Ex: Shift Registers) Application involving complementation (Ex: Binary Counters)

3.11.2 Excitation Tables:

Before going to the design examples for the clocked synchronous sequential circuits we revise Flip-Flop excitation tables.

Present State	Next State	Inputs	
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Excitation table for SR Flip-Flop

Present State	Next State	Inputs	
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table for JK Flip-Flop

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

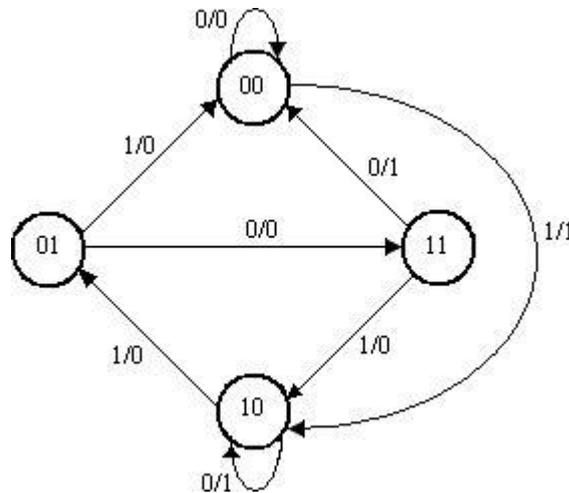
Excitation table for T Flip-Flop

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table for D Flip-Flop

3.11.3 Problems

1. A sequential circuit has one input and one output. The state diagram is shown below. Design the sequential circuit with a) D-Flip-Flops, b) T Flip-Flops, c) RS Flip-Flops and d) JK Flip-Flops.

**Solution:****State Table:**

The state table for the state diagram is,

Present state		Next state		Output	
		X= 0	X= 1	X= 0	X= 1
A	B	AB	AB	Y	Y
0	0	00	10	0	1
0	1	11	00	0	0
1	0	10	01	1	0
1	1	00	10	1	0

State reduction:

As seen from the state table there is no equivalent states. Therefore, no reduction in the state diagram.

The state table shows that circuit goes through four states, therefore we require 2 Flip-Flops (number of states= 2^m , where m = number of Flip-Flops). Since two Flip-Flops are required first is denoted as A and second is denoted as B.

i) Design using D Flip-Flops:

Excitation table:

Using the excitation table for T Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table for D Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs		Output
A	B	X	A	B	D_A	D_B	Y
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	1	1	1	1	0
0	1	1	0	0	0	0	0
1	0	0	1	0	1	0	1
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	1
1	1	1	1	0	1	0	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

$A \backslash BX$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$D_A = A'B'X + A'BX' + ABX + AB'X'$$

$$= A \oplus (B \oplus X)$$

For Flip-flop B

$A \backslash BX$	00	01	11	10
0	0	0	0	1
1	0	1	0	0

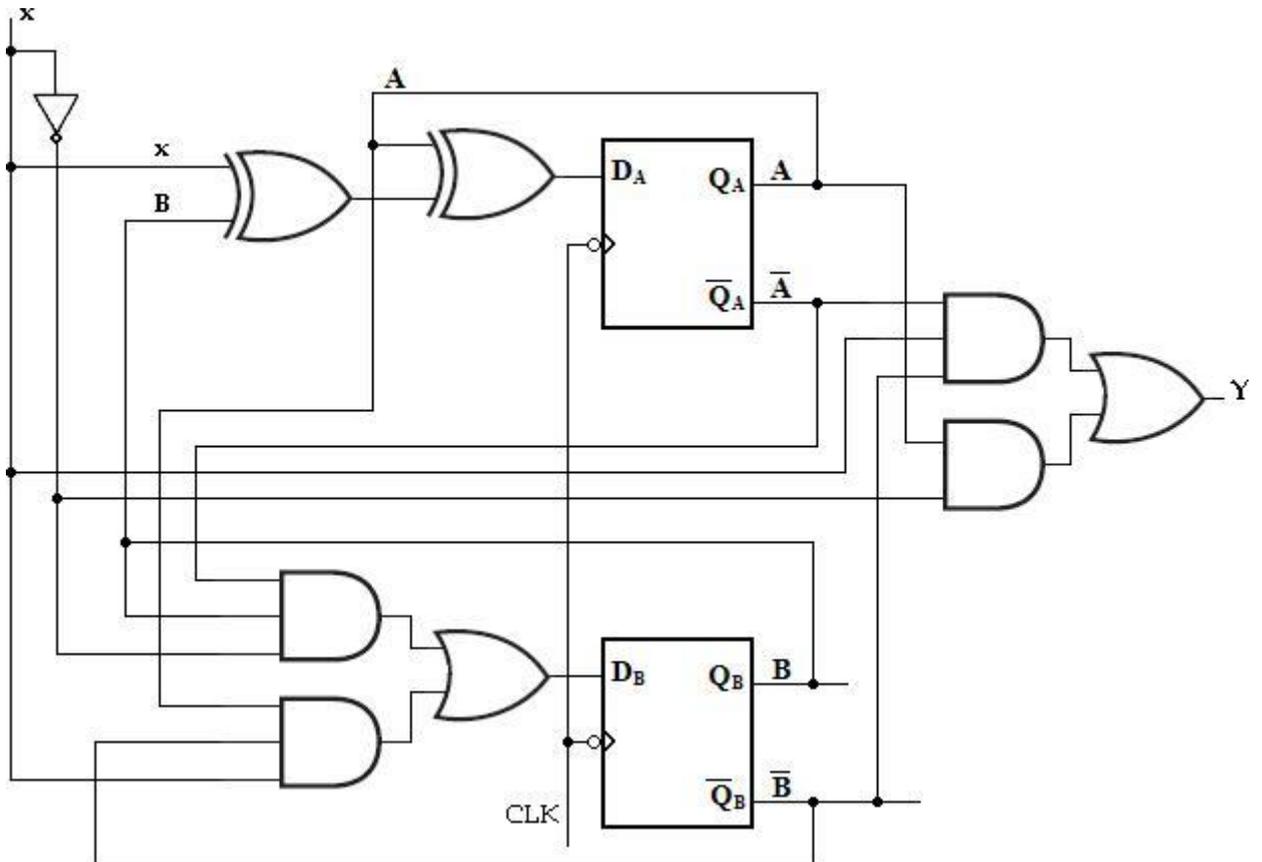
$$D_B = A'BX' + AB'X$$

For Output

$A \backslash BX$	00	01	11	10
0	0	1	0	0
1	1	0	0	1

$$Y = A'B'X + AX'$$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using D Flip-Flop

ii) Design using T Flip-Flops:

Using the excitation table for T Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Excitation table for T Flip-Flop

Present state		Input X	Next state		Flip-Flop Inputs		Output Y
A	B		A	B	T _A	T _B	
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	1	1	1	0	0
0	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1	1	0	0	1	0

Circuit excitation table

K-map Simplification:

		For Flip-flop A				For Flip-flop B				For Output			
		A \ BX				A \ BX				A \ BX			
		00	01	11	10	00	01	11	10	00	01	11	10
0	0	0	1	0	1	0	0	1	0	0	1	0	0
1	0	0	1	0	1	0	1	1	1	1	0	0	1
1	1	0	1	0	1	0	1	1	1	1	0	0	1

$T_A = B \oplus X$
 $T_B = AB + AX + BX$
 $Y = A'B'X + AX'$

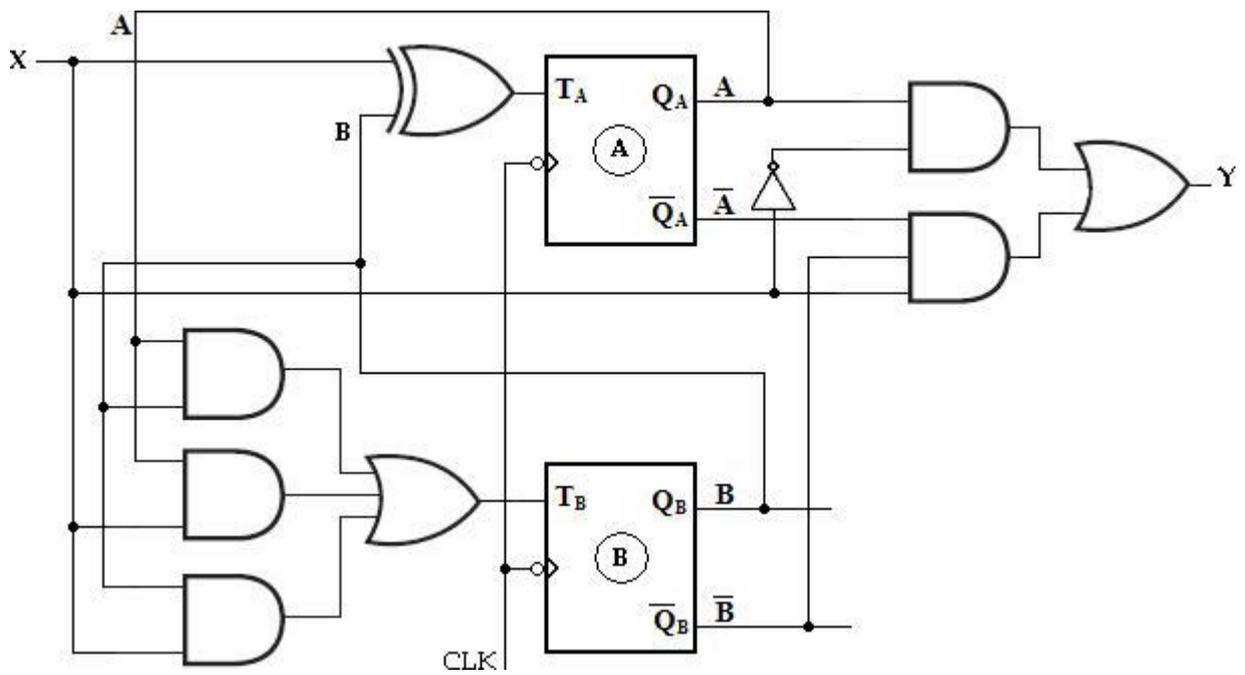
Therefore, input functions for,

$$T_A = B \oplus X \text{ and}$$

$$T_B = AB + AX + BX$$

Circuit output function, $Y = XA'B' + X'A$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using T Flip-Flop

iii) Design using SR Flip-Flops:

Using the excitation table for RS Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Inputs	
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Excitation table for SR Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs				Output
A	B	X	A	B	S_A	R_A	S_B	R_B	Y
0	0	0	0	0	0	x	0	x	0
0	0	1	1	0	1	0	0	x	1
0	1	0	1	1	1	0	x	0	0
0	1	1	0	0	0	x	0	1	0
1	0	0	1	0	x	0	0	x	1
1	0	1	0	1	0	1	1	0	0
1	1	0	0	0	0	1	0	1	1
1	1	1	1	0	x	0	0	1	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

For S_A

A \ BX	00	01	11	10
0	0	1	0	1
1	x	0	x	0

$$S_A = A'B'X + A'EX'$$

$$= A'(B \oplus X)$$

For R_A

A \ BX	00	01	11	10
0	x	0	x	0
1	0	1	0	1

$$R_A = ABX' + AB'X'$$

For Output

A \ BX	00	01	11	10
0	0	1	0	0
1	1	0	0	1

$$Y = A'B'X + AX'$$

For Flip-flop B

For S_B

A \ BX	00	01	11	10
0	0	0	0	x
1	0	1	0	0

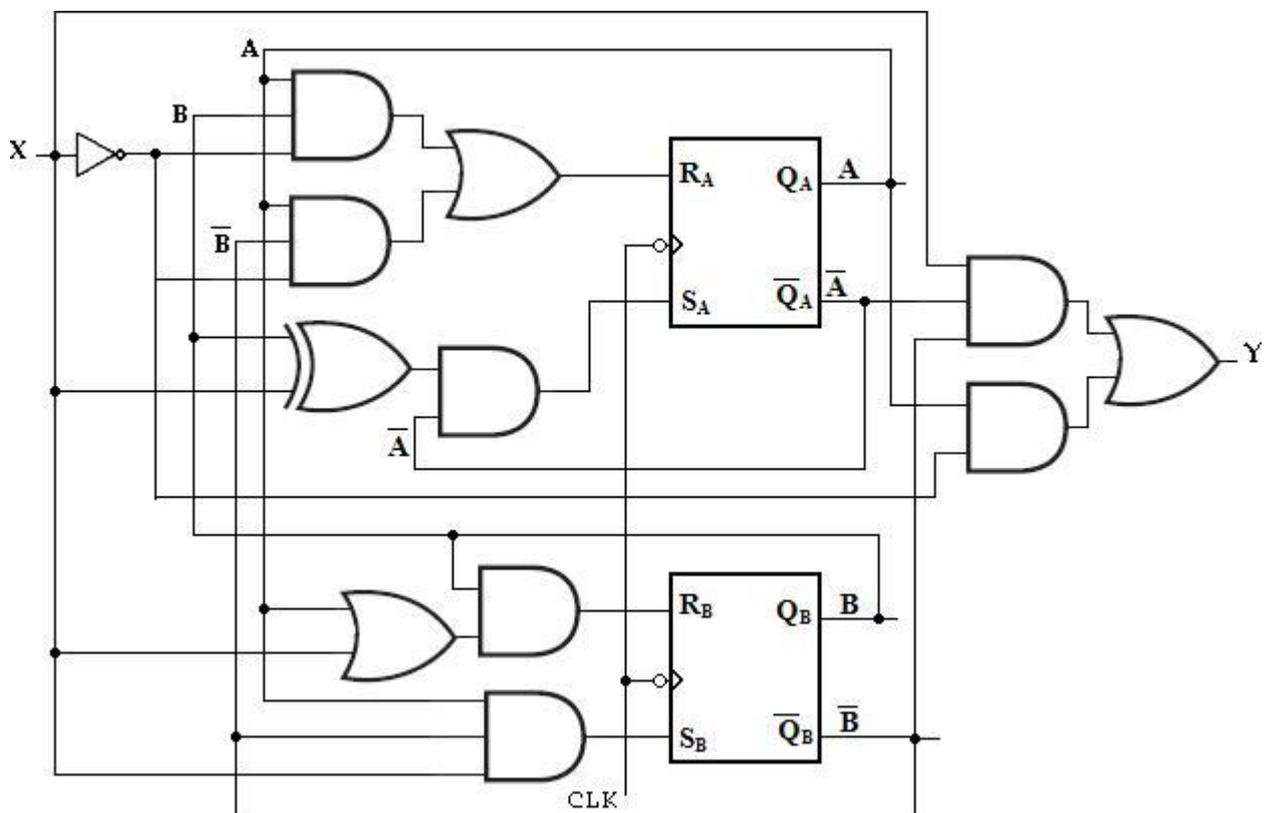
$$S_B = AB'X$$

For R_B

A \ BX	00	01	11	10
0	x	x	1	0
1	x	0	1	1

$$R_B = AB + BX$$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



iii) Design using JK Flip-Flops:

Using the excitation table for JK Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Inputs	
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table for JK Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs				Output
A	B	X	A	B	J_A	K_A	J_B	K_B	Y
0	0	0	0	0	0	x	0	x	0
0	0	1	1	0	1	x	0	x	1
0	1	0	1	1	1	x	x	0	0
0	1	1	0	0	0	x	x	1	0
1	0	0	1	0	x	0	0	x	1
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	1
1	1	1	1	0	x	0	x	1	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

For J_A

		BX			
A		00	01	11	10
0		0	1	0	1
1		x	x	x	x

$$J_A = BX' + B'X$$

$$= B \oplus X$$

For K_A

		BX			
A		00	01	11	10
0		x	x	x	x
1		0	1	0	1

$$K_A = BX' + B'X$$

$$= B \oplus X$$

For Output

		BX			
A		00	01	11	10
0		0	1	0	0
1		1	0	0	1

$$Y = A'B'X + AX'$$

For Flip-flop B

		For J_B			
		BX 00	01	11	10
A	0	0	0	x	x
	1	0	1	x	x

$J_B = AX$

		For K_B			
		BX 00	01	11	10
A	0	x	x	1	0
	1	x	x	1	1

$K_B = A + X$

The input functions for,

$$J_A = BX' + B'X$$

$$= B \uparrow X$$

$$J_B = AX$$

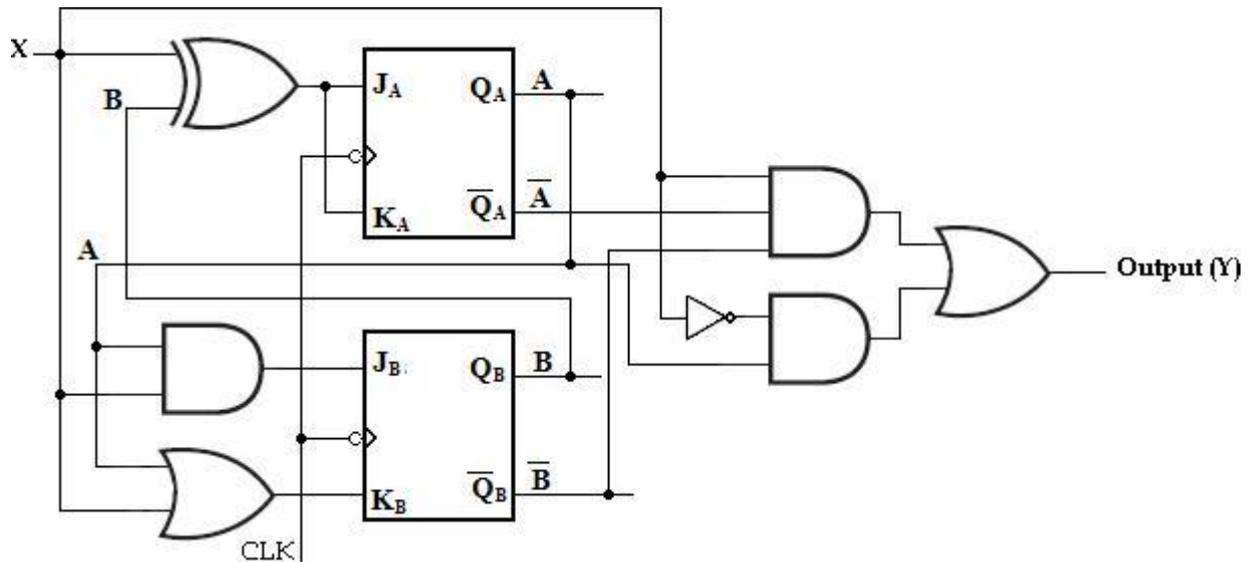
$$K_A = BX' + B'X$$

$$= B \uparrow X$$

$$K_B = A + X$$

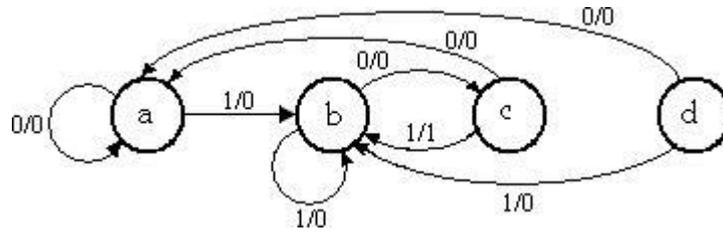
Circuit output function, $Y = AX' + A'B'X$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using JK Flip-Flop

2. Design a clocked sequential machine using JK Flip-Flops for the state diagram shown in the figure. Use state reduction if possible. Make proper state assignment.



Soln:

State Table:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	b	0	0
c	a	b	0	1
d	a	b	0	0

From the above state table **a** and **d** generate exactly same next state and same output for every possible set of inputs. The state **a** and **d** go to next states **a** and **b** and have outputs 0 and 0 for $x=0$ and $x=1$ respectively. Therefore state **d** can be removed and replaced by **a**. The final reduced state table is shown below.

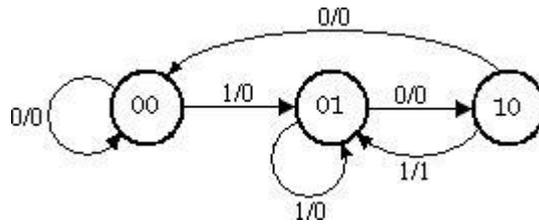
Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	b	0	0
c	a	b	0	1

Reduced State table

Binary Assignment:

Now each state is assigned with binary values. Since there are three states, number of Flip-Flops required is two and 2 binary numbers are assigned to the states. $a= 00$; $b= 01$; and $c= 10$

The reduced state diagram is drawn as,



Reduced State Diagram

Excitation Table:

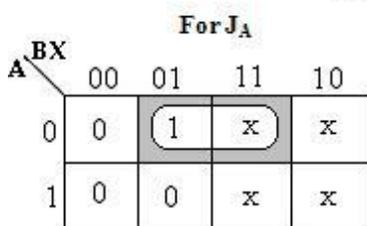
Present State	Next State	Inputs	
		J	K
Q_n	Q_{n+1}		
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table for JK Flip-Flop

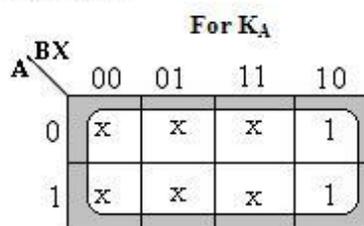
Input	Present state		Next state		Flip-Flop Inputs				Output
	X	A	B	A	B	J_A	K_A	J_B	
0	0	0	0	0	0	x	0	x	0
1	0	0	0	1	0	x	1	x	0
0	0	1	1	0	1	x	x	1	0
1	0	1	0	1	0	x	x	0	0
0	1	0	0	0	x	1	0	x	0
1	1	0	0	1	x	1	1	x	1
0	1	1	x	x	x	x	x	x	x
1	1	1	x	x	x	x	x	x	x

K-map Simplification:

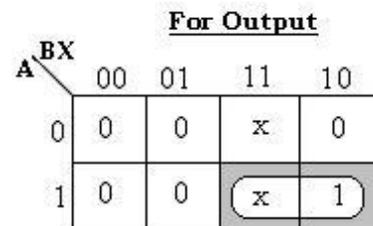
For Flip-flop A



$J_A = X'B$



$K_A = 1$



$Y = XA$

For Flip-flop B

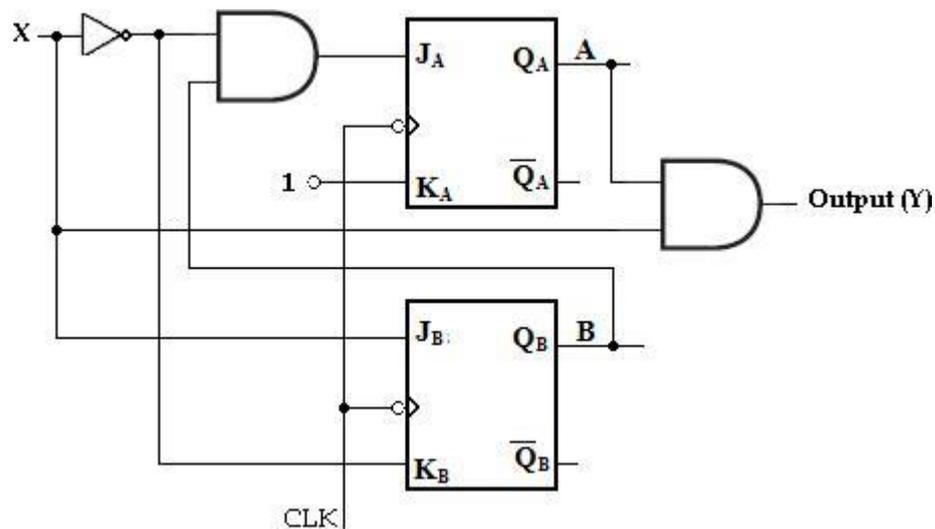
		For J_B			
A	BX	00	01	11	10
0		0	x	x	0
1		1	x	x	1

$J_B = X$

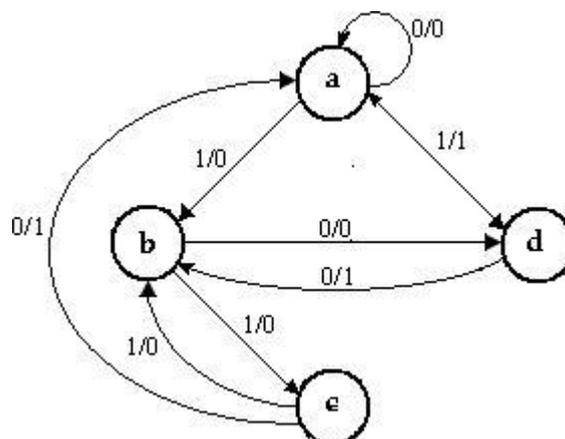
		For K_B			
A	BX	00	01	11	10
0		x	1	x	x
1		x	0	x	x

$K_B = X'$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



3. Design a clocked sequential machine using T Flip-Flops for the following state diagram. Use state reduction if possible. Also use straight binary state assignment.



Soln:

State Table:

State table for the given state diagram is,

Present state	Next state		Output	
	X=0	X=1	X=0	X=1
a	a	b	0	0
b	d	c	0	0
c	a	b	1	0
d	b	a	1	1

Even though a and c are having same next states for input X=0 and X=1, as the outputs are not same state reduction is not possible.

State Assignment:

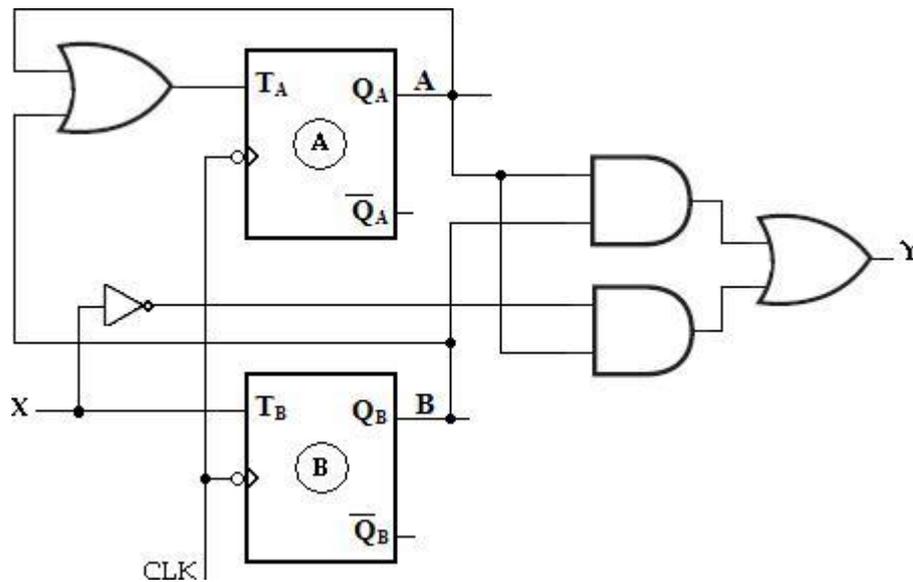
Use straight binary assignments as a= 00, b= 01, c= 10 and d= 11, the transition table is,

Input	Present state		Next state		Flip-Flop Inputs		Output
	X	A	B	A	B	T _A	
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1
0	1	1	0	1	1	0	1
1	0	0	0	1	0	1	0
1	0	1	1	0	1	1	0
1	1	0	0	1	1	1	0
1	1	1	0	0	1	1	1

K-map simplification:

For Flip-flop A					For Flip-flop B					For Output				
X\AB	00	01	11	10	X\AB	00	01	11	10	X\AB	00	01	11	10
0	0	1	1	1	0	0	0	0	0	0	0	0	1	1
1	0	1	1	1	1	1	1	1	1	1	0	0	1	0
$T_A = A + B$					$T_B = X$					$Z = AB + X'A$				

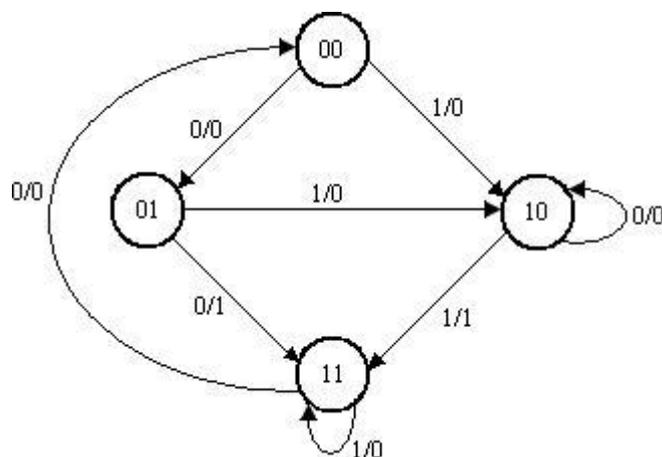
Logic Diagram:



3.12 STATE ASSIGNMENT:

In sequential circuits, the behavior of the circuit is defined in terms of its inputs, present states, next states and outputs. To generate desired next state at particular present state and inputs, it is necessary to have specific Flip-Flop inputs. These Flip-Flop inputs are described by a set of Boolean functions called Flip-Flop input functions.

To determine the Flip-Flop functions, it is necessary to represent states in the state diagram using binary values instead of alphabets. This procedure is known as *state assignment*.



Reduced state diagram with binary states

3.15.1 Rules for state assignments

There are two basic rules for making state assignments.

Rule 1:

States having the **same** NEXT STATES for a given input condition should have assignments which can be grouped into logically adjacent cells in a K-map.

Rule 2:

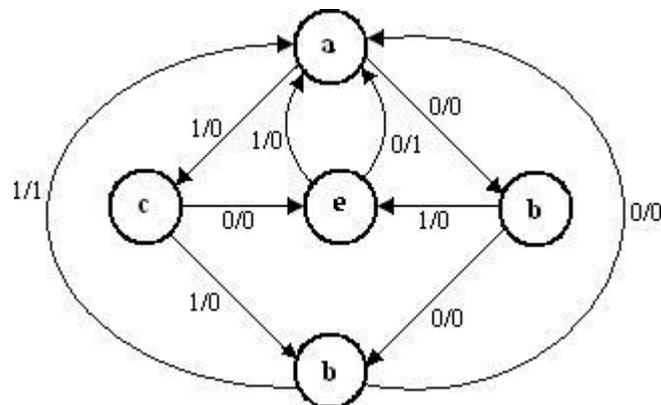
States that are the NEXT STATES of a single state should have assignment which can be grouped into logically adjacent cells in a K-map.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
00	01	10	0	0
01	11	10	1	0
10	10	11	0	1
11	00	11	0	0

State table with assignment states

3.15.2 State Assignment Problem:

1. Design a sequential circuit for a state diagram shown below. Use state assignment rules for assigning states and compare the required combinational circuit with random state assignment.



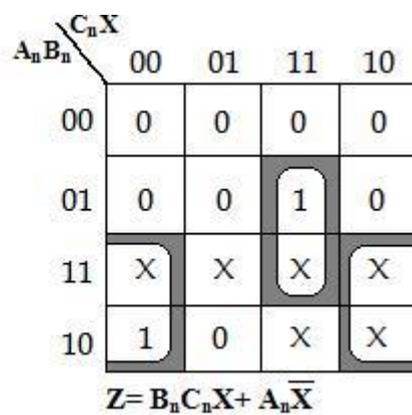
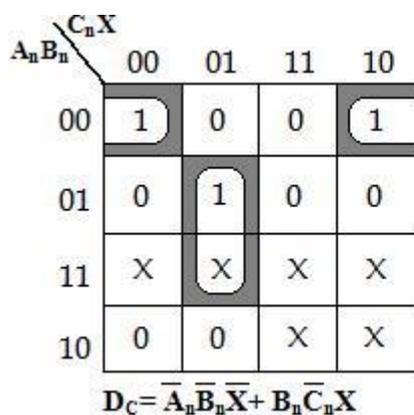
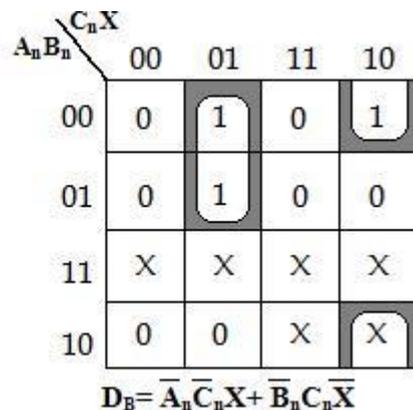
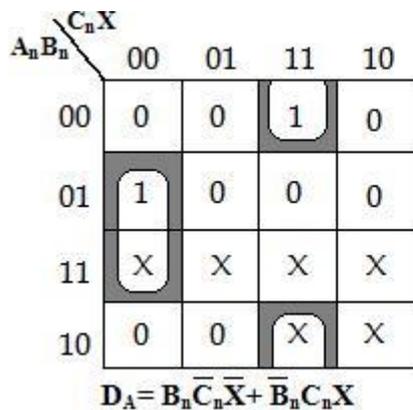
Using random state assignment we assign,

a= 000, b= 001, c= 010, d= 011 and e= 100.

The excitation table with these assignments is given as,

Present state			Input	Next state			Output
A _n	B _n	C _n	X	A _{n+1}	B _{n+1}	C _{n+1}	Z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	0	1	1	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

K-map Simplification:



The random assignments require:

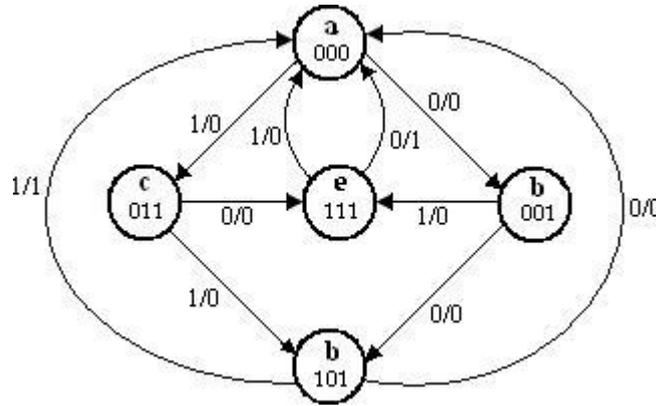
7 three input AND functions

1 two input AND function

4 two input OR functions

12 gates with 31 inputs

Now, we will apply the state assignment rules and compare the results.



State diagram after applying Rules 1 and 2

Rule 1 says that: e and d must be adjacent, and
b and c must be adjacent.

Rule 2 says that: e and d must be adjacent, and
b and c must be adjacent.

Applying Rule 1, Rule 2 to the state diagram we get the state assignment as,

Present state			Input	Next state			Output
A _n	B _n	C _n	X	A _{n+1}	B _{n+1}	C _{n+1}	Z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0
0	0	1	0	1	0	1	0
0	0	1	1	1	1	1	0
0	1	0	0	x	x	x	x
0	1	0	1	x	x	x	x
0	1	1	0	1	1	1	0
0	1	1	1	1	0	1	0
1	0	0	0	x	x	x	x
1	0	0	1	x	x	x	x
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

K-map Simplification:

$A_n B_n \backslash C_n X$	00	01	11	10
00	0	0	1	1
01	X	X	1	1
11	X	X	0	0
10	X	X	0	0

$A_{n+1} = D_A = \bar{A}_n C_n$

$A_n B_n \backslash C_n X$	00	01	11	10
00	0	1	1	0
01	X	X	0	1
11	X	X	0	0
10	X	X	0	0

$B_{n+1} = D_B = \bar{A}_n \bar{B}_n X + \bar{A}_n B_n \bar{X}$

$A_n B_n \backslash C_n X$	00	01	11	10
00	1	1	1	1
01	X	X	1	1
11	X	X	0	0
10	X	X	0	0

$C_{n+1} = D_C = \bar{A}_n$

$A_n B_n \backslash C_n X$	00	01	11	10
00	0	0	1	0
01	X	X	0	0
11	X	X	0	1
10	X	X	1	0

$Z = A_n B_n \bar{X} + A_n \bar{B}_n X$

The state assignments using Rule 1 and 2 require:

- 4 three input AND functions
- 1 two input AND function
- 2 two input OR functions

7 gates with 18 inputs

Thus by simply applying Rules 1 and 2 good results have been achieved.

3.14 SYNCHRONOUS COUNTERS

Flip-Flops can be connected together to perform counting operations. Such a group of Flip-Flops is a **counter**. The number of Flip-Flops used and the way in which they are connected determine the number of states (called the modulus) and also the specific sequence of states that the counter goes through during each complete cycle.

Counters are classified into two broad categories according to the way they are clocked:

- ✚ Asynchronous counters,
- ✚ Synchronous counters.

In asynchronous (ripple) counters, the first Flip-Flop is clocked by the external clock pulse and then each successive Flip-Flop is clocked by the output of the preceding Flip-Flop.

In synchronous counters, the clock input is connected to all of the Flip-Flops so that they are clocked simultaneously. Within each of these two categories, counters are classified primarily by the type of sequence, the number of states, or the number of Flip-Flops in the counter.

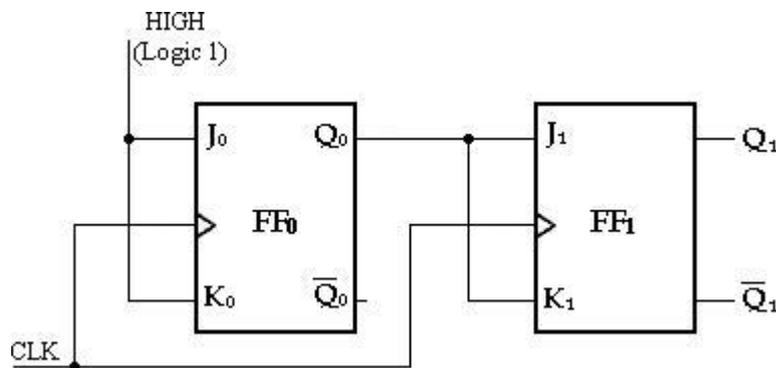
The term 'synchronous' refers to events that have a fixed time relationship with each other. In synchronous counter, the clock pulses are applied to all Flip-Flops simultaneously. Hence there is minimum propagation delay.

S.No	Asynchronous (ripple) counter	Synchronous counter
1	All the Flip-Flops are not clocked simultaneously.	All the Flip-Flops are clocked simultaneously.
2	The delay times of all Flip-Flops are added. Therefore there is considerable propagation delay.	There is minimum propagation delay.
3	Speed of operation is low	Speed of operation is high.
4	Logic circuit is very simple	Design involves complex logic circuit

	even for more number of states.	as number of state increases.
5	Minimum numbers of logic devices are needed.	The number of logic devices is more than ripple counters.
6	Cheaper than synchronous counters.	Costlier than ripple counters.

3.14.1 2-Bit Synchronous Binary Counter

In this counter the clock signal is connected in parallel to clock inputs of both the Flip-Flops (FF₀ and FF₁). The output of FF₀ is connected to J₁ and K₁ inputs of the second Flip-Flop (FF₁).



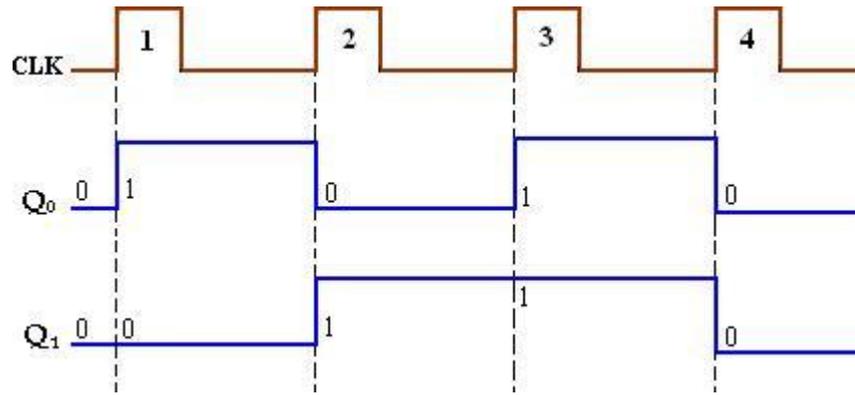
2-Bit Synchronous Binary Counter

Assume that the counter is initially in the binary 0 state: i.e., both Flip-Flops are RESET. When the positive edge of the first clock pulse is applied, FF₀ will toggle because $J_0 = K_0 = 1$, whereas FF₁ output will remain 0 because $J_1 = K_1 = 0$. After the first clock pulse $Q_0 = 1$ and $Q_1 = 0$.

When the leading edge of CLK₂ occurs, FF₀ will toggle and Q_0 will go LOW. Since FF₁ has a HIGH ($Q_0 = 1$) on its J₁ and K₁ inputs at the triggering edge of this clock pulse, the Flip-Flop toggles and Q_1 goes HIGH. Thus, after CLK₂, $Q_0 = 0$ and $Q_1 = 1$.

When the leading edge of CLK₃ occurs, FF₀ again toggles to the SET state ($Q_0 = 1$), and FF₁ remains SET ($Q_1 = 1$) because its J₁ and K₁ inputs are both LOW ($Q_0 = 0$). After this triggering edge, $Q_0 = 1$ and $Q_1 = 1$.

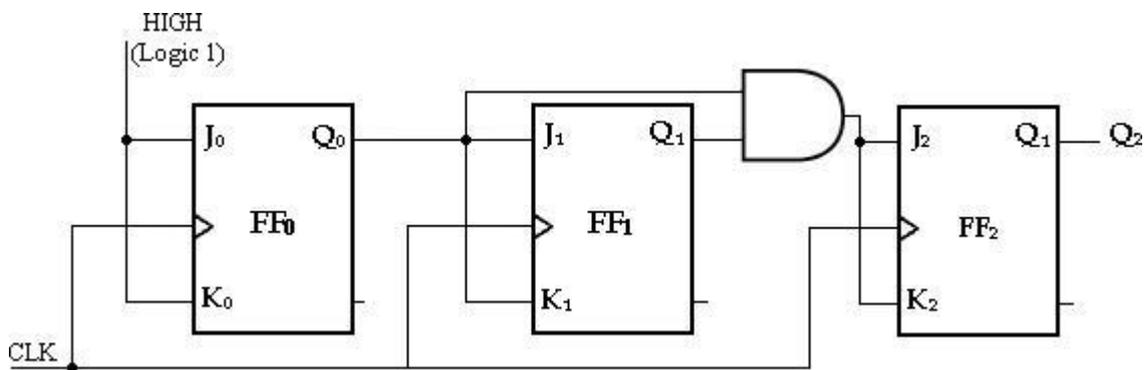
Finally, at the leading edge of CLK₄, Q_0 and Q_1 go LOW because they both have a toggle condition on their J₁ and K₁ inputs. The counter has now recycled to its original state, $Q_0 = Q_1 = 0$.



Timing diagram

3.14.2 3-Bit Synchronous Binary Counter

A 3 bit synchronous binary counter is constructed with three JK Flip-Flops and an AND gate. The output of FF₀ (Q₀) changes on each clock pulse as the counter progresses from its original state to its final state and then back to its original state. To produce this operation, FF₀ must be held in the toggle mode by constant HIGH, on its J₀ and K₀ inputs.



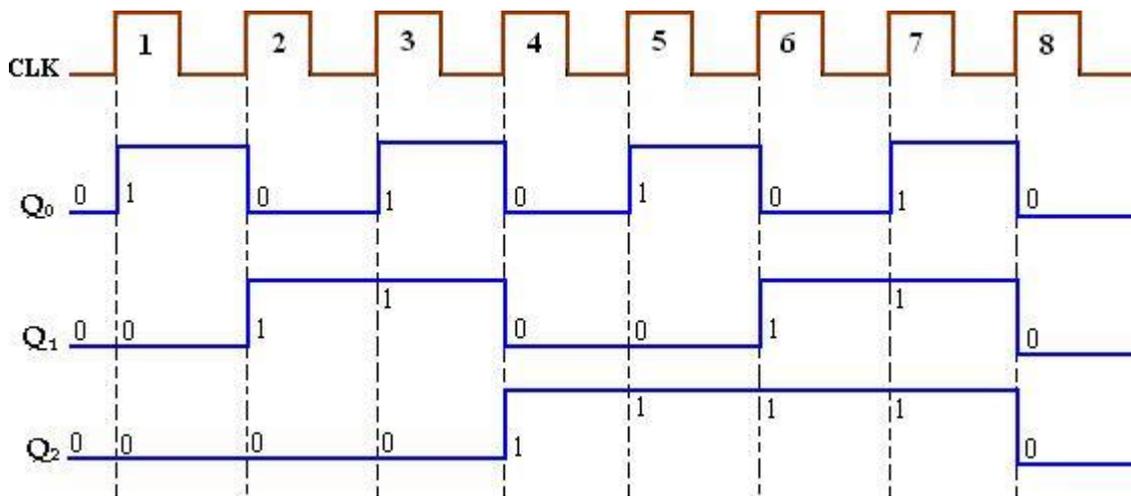
3-Bit Synchronous Binary Counter

The output of FF₁ (Q₁) goes to the opposite state following each time Q₀ = 1. This change occurs at CLK₂, CLK₄, CLK₆, and CLK₈. The CLK₈ pulse causes the counter to recycle. To produce this operation, Q₀ is connected to the J₁ and K₁ inputs of FF₁. When Q₀ = 1 and a clock pulse occurs, FF₁ is in the toggle mode and therefore changes state. When Q₀ = 0, FF₁ is in the no-change mode and remains in its present state.

The output of FF₂ (Q₂) changes state both times; it is preceded by the unique condition in which both Q₀ and Q₁ are HIGH. This condition is detected by the AND gate and applied to the J₂ and K₂ inputs of FF₃. Whenever both outputs Q₀ = Q₁ = 1,

the output of the AND gate makes the $J_2= K_2= 1$ and FF2 toggles on the following clock pulse. Otherwise, the J_2 and K_2 inputs of FF2 are held LOW by the AND gate output, FF2 does not change state.

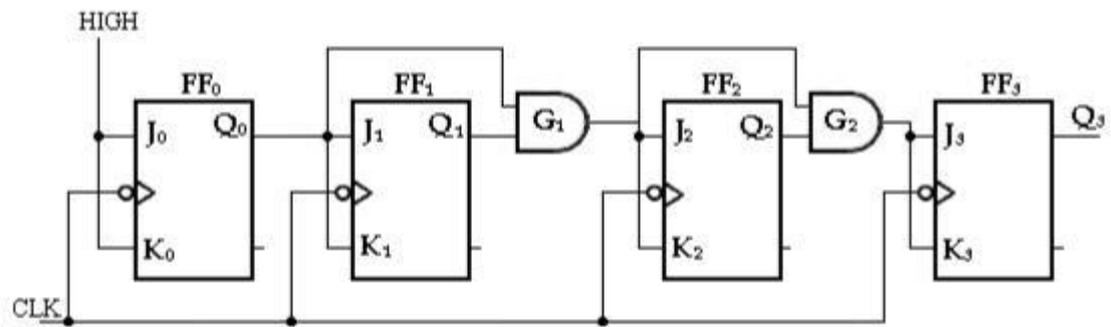
CLOCK Pulse	Q ₂	Q ₁	Q ₀
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0



Timing diagram

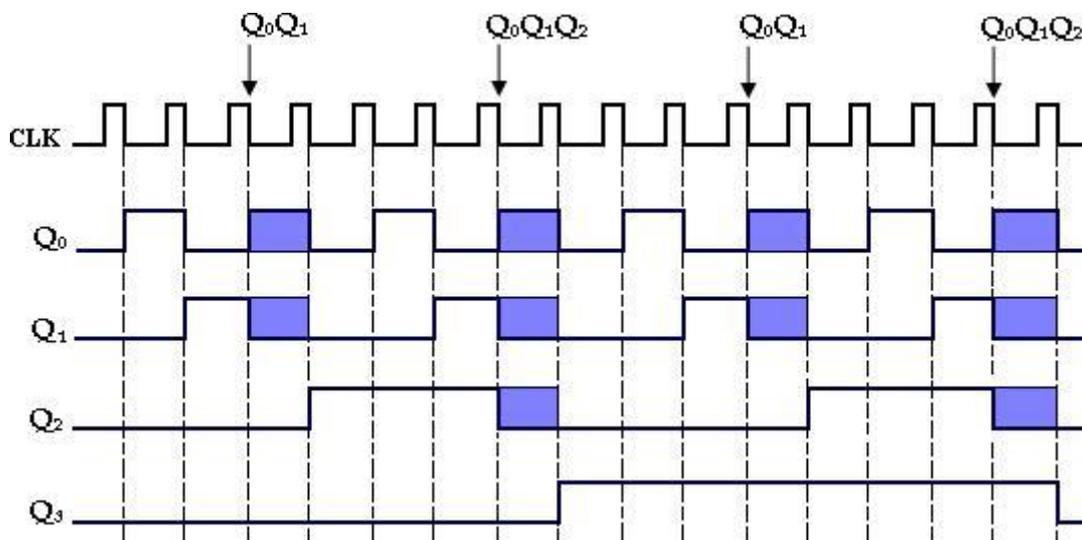
3.14.3 4-Bit Synchronous Binary Counter

This particular counter is implemented with negative edge-triggered Flip-Flops. The reasoning behind the J and K input control for the first three Flip-Flops is the same as previously discussed for the 3-bit counter. For the fourth stage, the Flip-Flop has to change the state when $Q_0= Q_1= Q_2= 1$. This condition is decoded by AND gate G₃.



4-Bit Synchronous Binary Counter

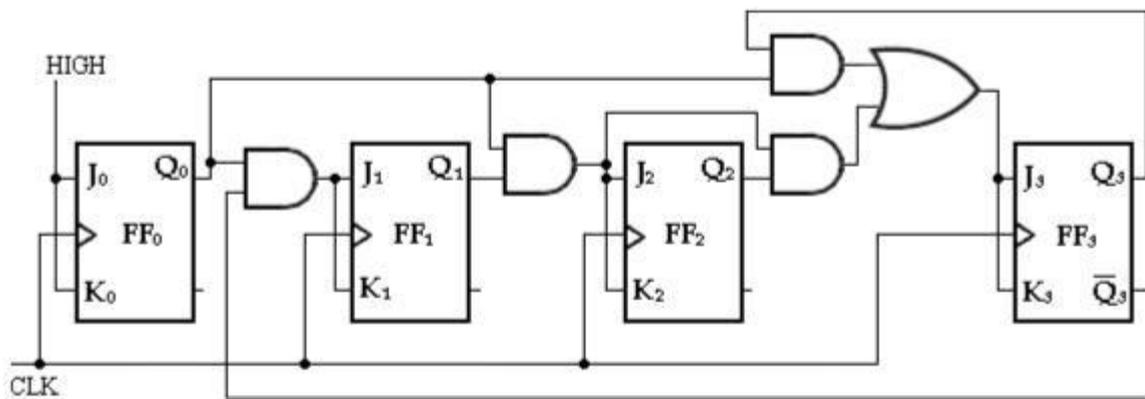
Therefore, when $Q_0 = Q_1 = Q_2 = 1$, Flip-Flop FF₃ toggles and for all other times it is in a no-change condition. Points where the AND gate outputs are HIGH are indicated by the shaded areas.



Timing diagram

3.14.4 4-Bit Synchronous Decade Counter: (BCD Counter):

BCD decade counter has a sequence from 0000 to 1001 (9). After 1001 state it must recycle back to 0000 state. This counter requires four Flip-Flops and AND/OR logic as shown below.



4-Bit Synchronous Decade Counter

CLOCK Pulse	Q ₃	Q ₂	Q ₁	Q ₀
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10(recycles)	0	0	0	0

- x First, notice that FF₀ (Q₀) toggles on each clock pulse, so the logic equation for its J₀ and K₀ inputs is

$$J_0 = K_0 = 1$$

This equation is implemented by connecting J₀ and K₀ to a constant HIGH level.

- x Next, notice from table, that FF₁ (Q₁) changes on the next clock pulse each time Q₀ = 1 and Q₃ = 0, so the logic equation for the J₁ and K₁ inputs is

$$J_1 = K_1 = Q_0 Q_3'$$

This equation is implemented by ANDing Q₀ and Q₃ and connecting the gate output to the J₁ and K₁ inputs of FF₁.

- x Flip-Flop 2 (Q₂) changes on the next clock pulse each time both Q₀ = Q₁ = 1. This requires an input logic equation as follows:

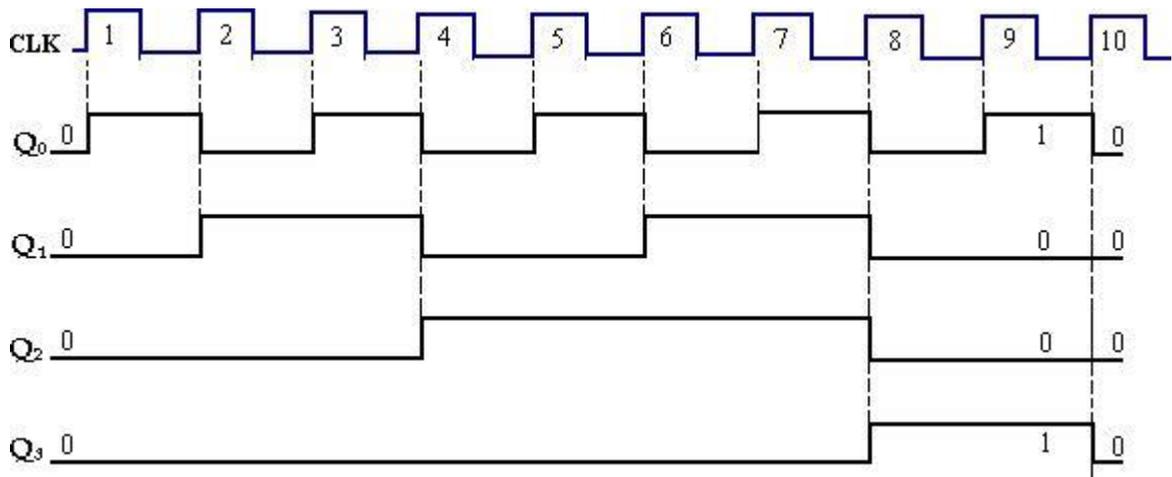
$$J_2 = K_2 = Q_0 Q_1$$

This equation is implemented by ANDing Q₀ and Q₁ and connecting the gate output to the J₂ and K₂ inputs of FF₂.

- x Finally, FF₃ (Q₃) changes to the opposite state on the next clock pulse each time Q₀ = 1, Q₁ = 1, and Q₂ = 1 (state 7), or when Q₀ = 1 and Q₁ = 1 (state 9). The equation for this is as follows:

$$J_3 = K_3 = Q_0 Q_1 Q_2 + Q_0 Q_3$$

This function is implemented with the AND/OR logic connected to the J₃ and K₃ inputs of FF₃.



Timing diagram

3.14.5 Synchronous UP/DOWN Counter

An up/down counter is a bidirectional counter, capable of progressing in either direction through a certain sequence. A 3-bit binary counter that advances upward through its sequence (0, 1, 2, 3, 4, 5, 6, 7) and then can be reversed so that it goes through the sequence in the opposite direction (7, 6, 5, 4, 3, 2, 1, 0) is an illustration of up/down sequential operation.

The complete up/down sequence for a 3-bit binary counter is shown in table below. The arrows indicate the state-to-state movement of the counter for both its UP and its DOWN modes of operation. An examination of Q₀ for both the up and down sequences shows that FF₀ toggles on each clock pulse. Thus, the J₀ and K₀ inputs of FF₀ are,

$$J_0 = K_0 = 1$$

CLOCK PULSE	UP	Q ₂	Q ₁	Q ₀	DOWN
0	↶	0	0	0	↷
1	↶	0	0	1	↷
2	↶	0	1	0	↷
3	↶	0	1	1	↷
4	↶	1	0	0	↷
5	↶	1	0	1	↷
6	↶	1	1	0	↷
7	↶	1	1	1	↷

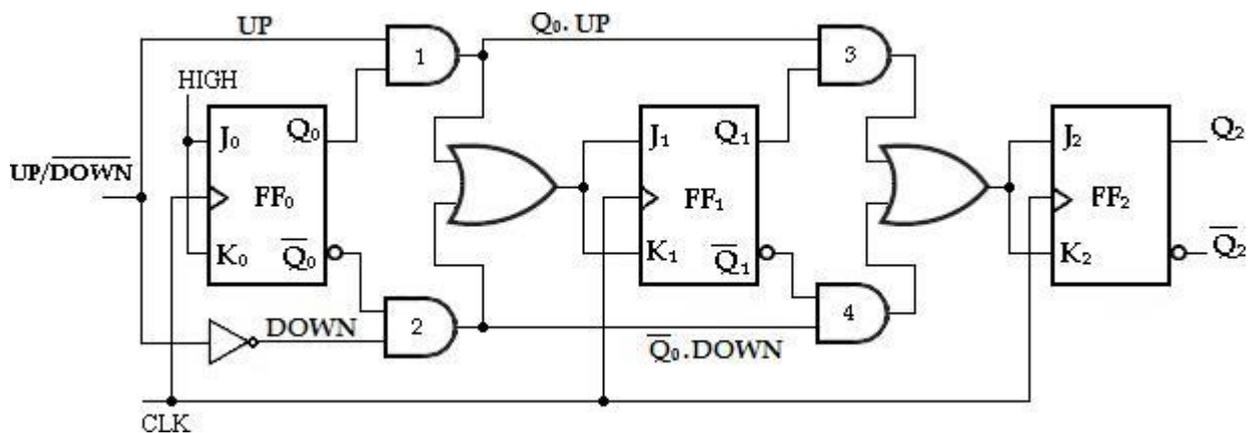
To form a synchronous UP/DOWN counter, the control input (UP/DOWN) is used to allow either the normal output or the inverted output of one Flip-Flop to the J and K inputs of the next Flip-Flop. When UP/DOWN= 1, the MOD 8 counter will count from 000 to 111 and UP/DOWN= 0, it will count from 111 to 000.

When UP/DOWN= 1, it will enable AND gates 1 and 3 and disable AND gates 2 and 4. This allows the Q₀ and Q₁ outputs through the AND gates to the J and K inputs of the following Flip-Flops, so the counter counts up as pulses are applied.

When UP/DOWN= 0, the reverse action takes place.

$$J_1 = K_1 = (Q_0 \cdot UP) + (Q_0' \cdot DOWN)$$

$$J_2 = K_2 = (Q_0 \cdot Q_1 \cdot UP) + (Q_0' \cdot Q_1' \cdot DOWN)$$



3-bit UP/DOWN Synchronous Counter

3.14.6 MODULUS-N-COUNTERS

The counter with 'n' Flip-Flops has maximum MOD number 2^n . Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

$$2^n \geq N$$

- (i) For example, a 3 bit binary counter is a MOD 8 counter. The basic counter can be modified to produce MOD numbers less than 2^n by allowing the counter to skip those are normally part of counting sequence.

$$n = 3$$

$$N = 8$$

$$2^n = 2^3 = 8 = N$$

- (ii) **MOD 5 Counter:**

$$2^n = N$$

$$2^n = 5$$

$$2^2 = 4 \text{ less than } N.$$

$$2^3 = 8 > N(5)$$

Therefore, 3 Flip-Flops are required.

- (iii) **MOD 10 Counter:**

$$2^n = N = 10$$

$$2^3 = 8 \text{ less than } N.$$

$$2^4 = 16 > N(10).$$

To construct any MOD-N counter, the following methods can be used.

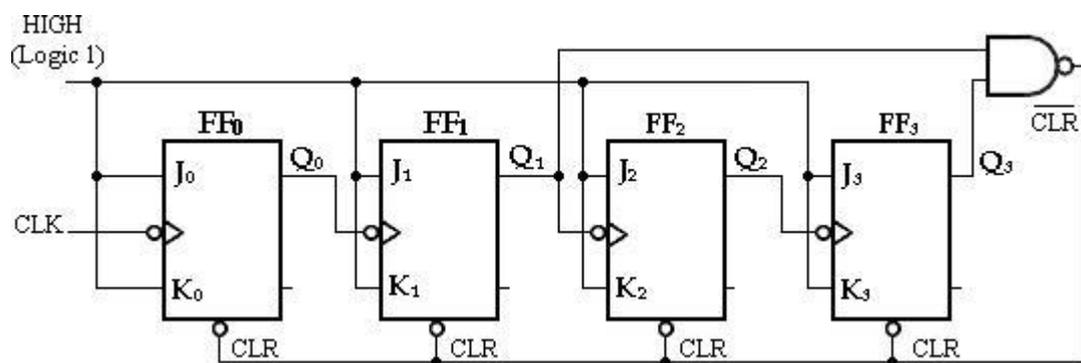
1. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

$$2^n \geq N.$$
2. Connect all the Flip-Flops as a required counter.
3. Find the binary number for N.
4. Connect all Flip-Flop outputs for which Q= 1 when the count is N, as inputs to NAND gate.
5. Connect the NAND gate output to the CLR input of each Flip-Flop.

When the counter reaches N^{th} state, the output of the NAND gate goes LOW, resetting all Flip-Flops to 0. Therefore the counter counts from 0 through $N-1$.

For example, MOD-10 counter reaches state 10 (1010). i.e., $Q_3Q_2Q_1Q_0 = 1\ 0\ 1\ 0$. The outputs Q_3 and Q_1 are connected to the NAND gate and the output of the NAND gate goes LOW and resetting all Flip-Flops to zero. Therefore MOD-10 counter counts from 0000 to 1001. And then recycles to the zero value.

The MOD-10 counter circuit is shown below.



MOD-10 (Decade) Counter

3.15 SHIFT REGISTERS:

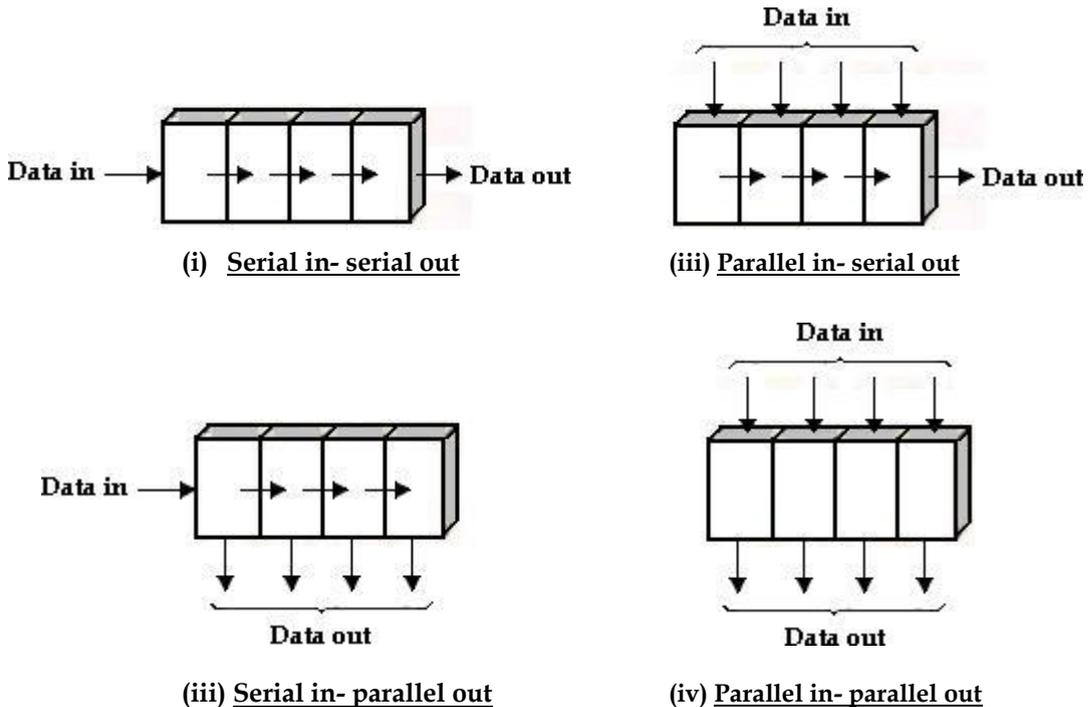
A register is simply a group of Flip-Flops that can be used to store a binary number. There must be one Flip-Flop for each bit in the binary number. For instance, a register used to store an 8-bit binary number must have 8 Flip-Flops.

The Flip-Flops must be connected such that the binary number can be entered (shifted) into the register and possibly shifted out. A group of Flip-Flops connected to provide either or both of these functions is called a *shift register*.

The bits in a binary number (data) can be removed from one place to another in either of two ways. The first method involves shifting the data one bit at a time in a serial fashion, beginning with either the most significant bit (MSB) or the least significant bit (LSB). This technique is referred to as *serial shifting*. The second method involves shifting all the data bits simultaneously and is referred to as *parallel shifting*.

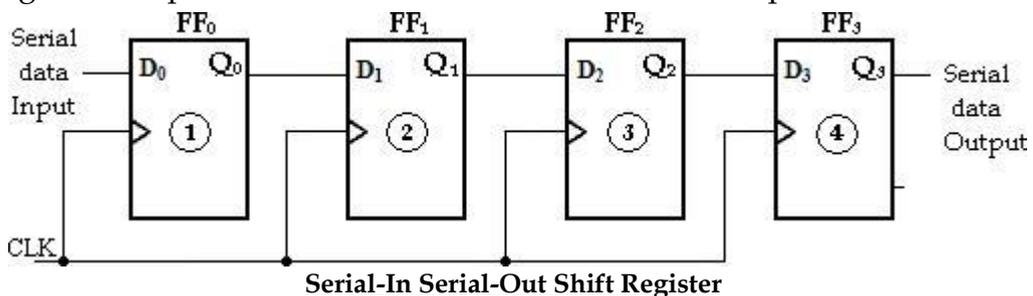
There are two ways to shift into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic register types –

- i. Serial in- serial out,
- ii. Serial in- parallel out,
- iii. Parallel in- serial out,
- iv. Parallel in- parallel out.



3.15.1 Serial-In Serial-Out Shift Register:

The serial in/serial out shift register accepts data serially, i.e., one bit at a time on a single line. It produces the stored information on its output also in serial form.



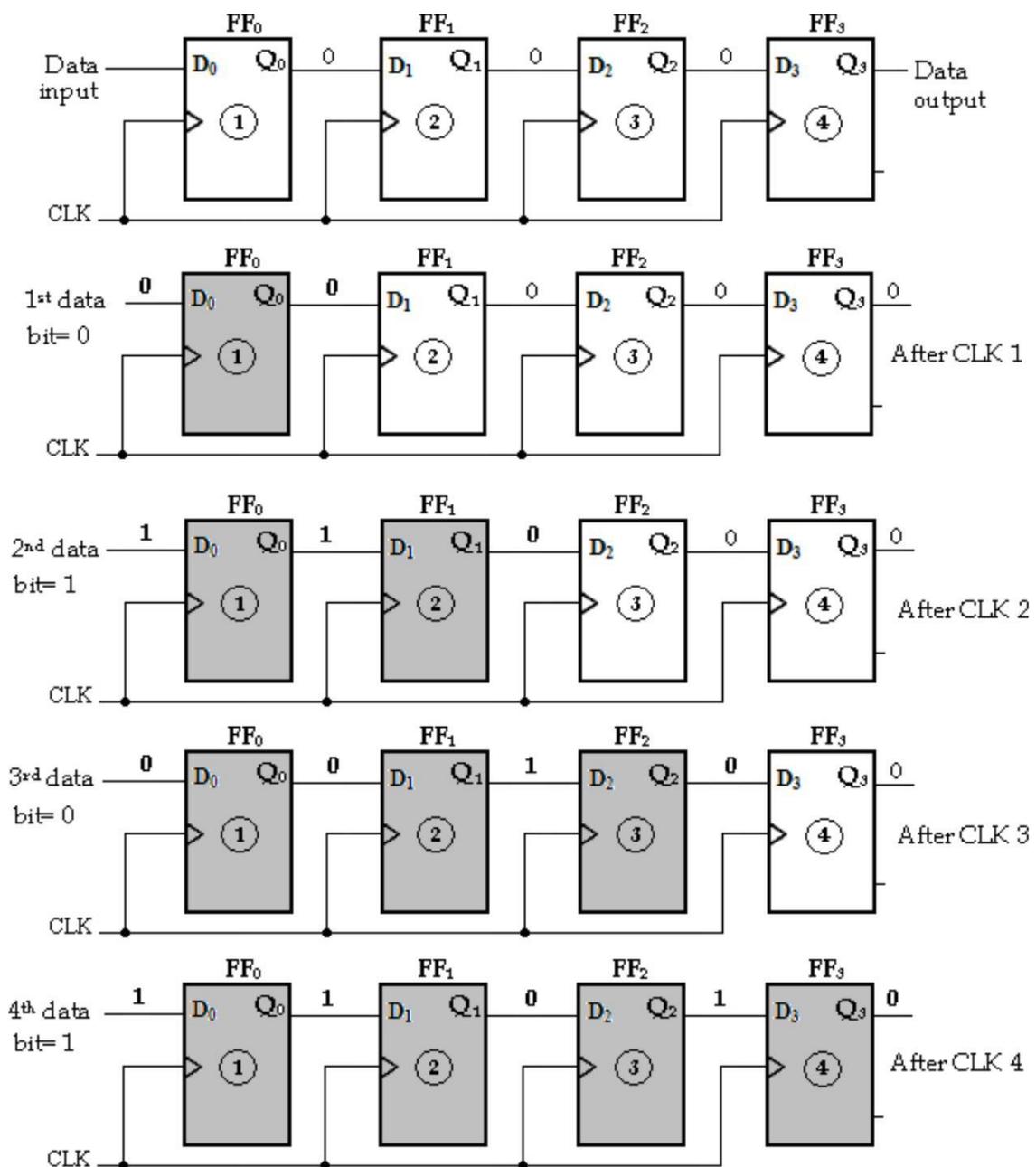
The entry of the four bits 1010 into the register is illustrated below, beginning with the right-most bit. The register is initially clear. The 0 is put onto the data input line, making $D=0$ for FF₀. When the first clock pulse is applied, FF₀ is reset, thus storing the 0.

Next the second bit, which is a 1, is applied to the data input, making $D=1$ for FF₀ and $D=0$ for FF₁ because the D input of FF₁ is connected to the Q₀ output. When

the second clock pulse occurs, the 1 on the data input is shifted into FF₀, causing FF₀ to set; and the 0 that was in FF₀ is shifted into FF₁.

The third bit, a 0, is now put onto the data-input line, and a clock pulse is applied. The 0 is entered into FF₀, the 1 stored in FF₀ is shifted into FF₁, and the 0 stored in FF₁ is shifted into FF₂.

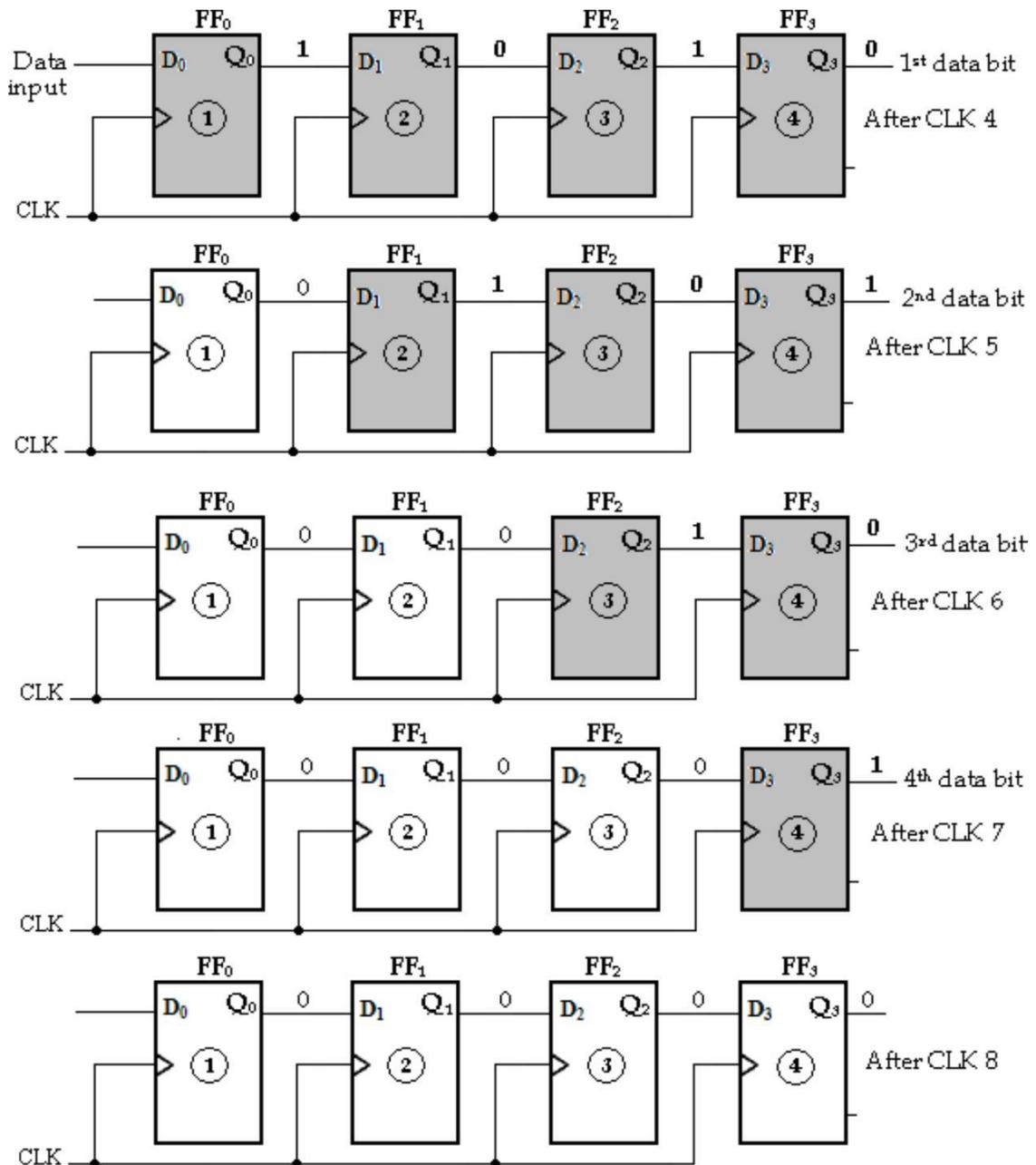
The last bit, a 1, is now applied to the data input, and a clock pulse is applied. This time the 1 is entered into FF₀, the 0 stored in FF₀ is shifted into FF₁, the 1 stored in FF₁ is shifted into FF₂, and the 0 stored in FF₂ is shifted into FF₃. This completes the serial entry of the four bits into the shift register, where they can be stored for any length of time as long as the Flip-Flops have dc power.



Four bits (1010) being entered serially into the register

To get the data out of the register, the bits must be shifted out serially and taken off the Q3 output. After CLK4, the right-most bit, 0, appears on the Q3 output.

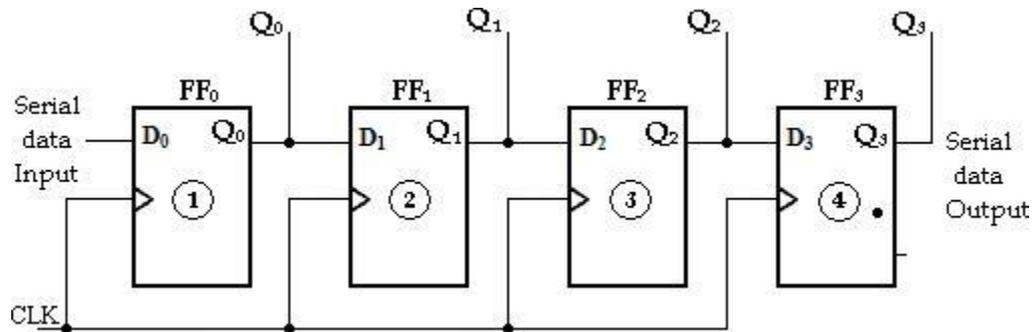
When clock pulse CLK5 is applied, the second bit appears on the Q3 output. Clock pulse CLK6 shifts the third bit to the output, and CLK7 shifts the fourth bit to the output. While the original four bits are being shifted out, more bits can be shifted in. All zeros are shown being shifted out, more bits can be shifted in.



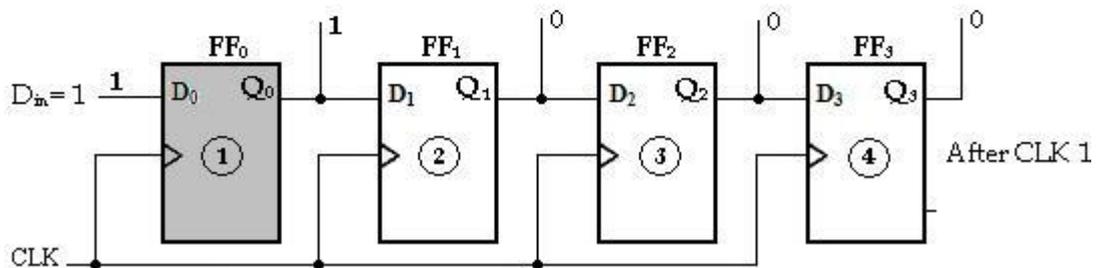
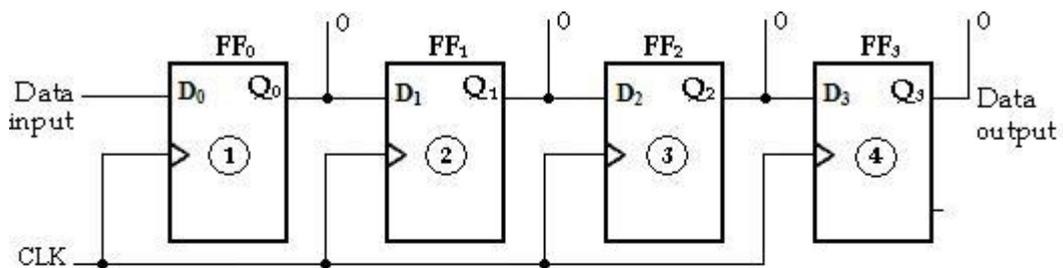
Four bits (1010) being entered serially-shifted out of the register and replaced by all zeros

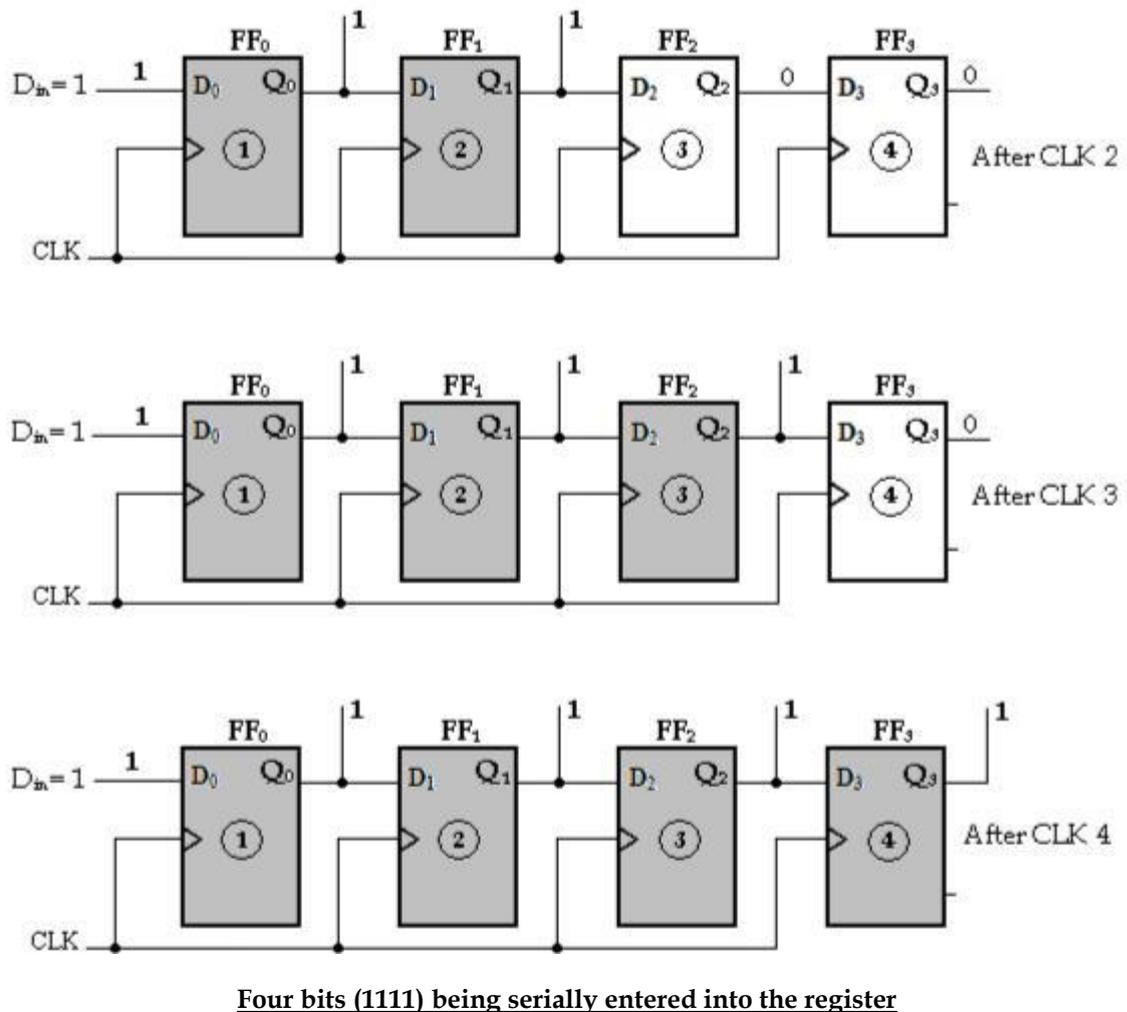
3.15.2 Serial-In Parallel-Out Shift Register:

In this shift register, data bits are entered into the register in the same as serial-in serial-out shift register. But the output is taken in parallel. Once the data are stored, each bit appears on its respective output line and all bits are available simultaneously instead of on a bit-by-bit.



Serial-In parallel-Out Shift Register



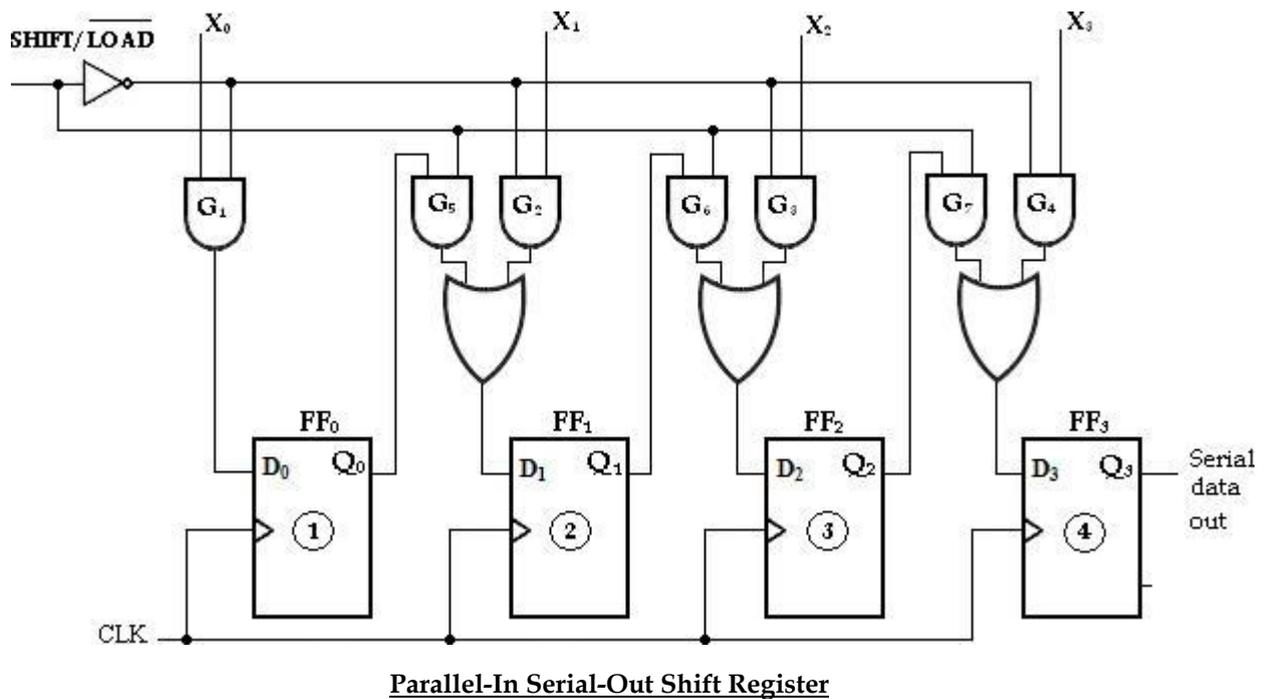


3.15.3 Parallel-In Serial-Out Shift Register:

In this type, the bits are entered in parallel i.e., simultaneously into their respective stages on parallel lines.

A 4-bit parallel-in serial-out shift register is illustrated below. There are four data input lines, X_0 , X_1 , X_2 and X_3 for entering data in parallel into the register. SHIFT/LOAD input is the control input, which allows four bits of data to **load** in parallel into the register.

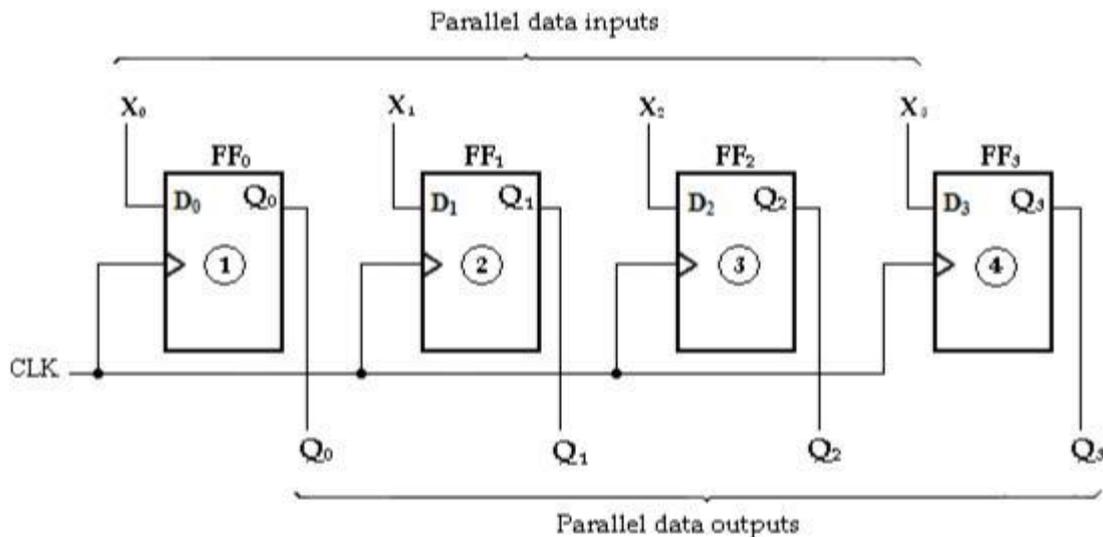
When SHIFT/LOAD is LOW, gates G_1 , G_2 , G_3 and G_4 are enabled, allowing each data bit to be applied to the D input of its respective Flip-Flop. When a clock pulse is applied, the Flip-Flops with $D = 1$ will **set** and those with $D = 0$ will **reset**, thereby storing all four bits simultaneously.



When SHIFT/LOAD is HIGH, gates G_1 , G_2 , G_3 and G_4 are disabled and gates G_5 , G_6 and G_7 are enabled, allowing the data bits to shift right from one stage to the next. The OR gates allow either the normal shifting operation or the parallel data-entry operation, depending on which AND gates are enabled by the level on the SHIFT/LOAD input.

3.15.4 Parallel-In Parallel-Out Shift Register:

In this type, there is simultaneous entry of all data bits and the bits appear on parallel outputs simultaneously.



Parallel-In Parallel-Out Shift Register

3.15.5 UNIVERSAL SHIFT REGISTERS

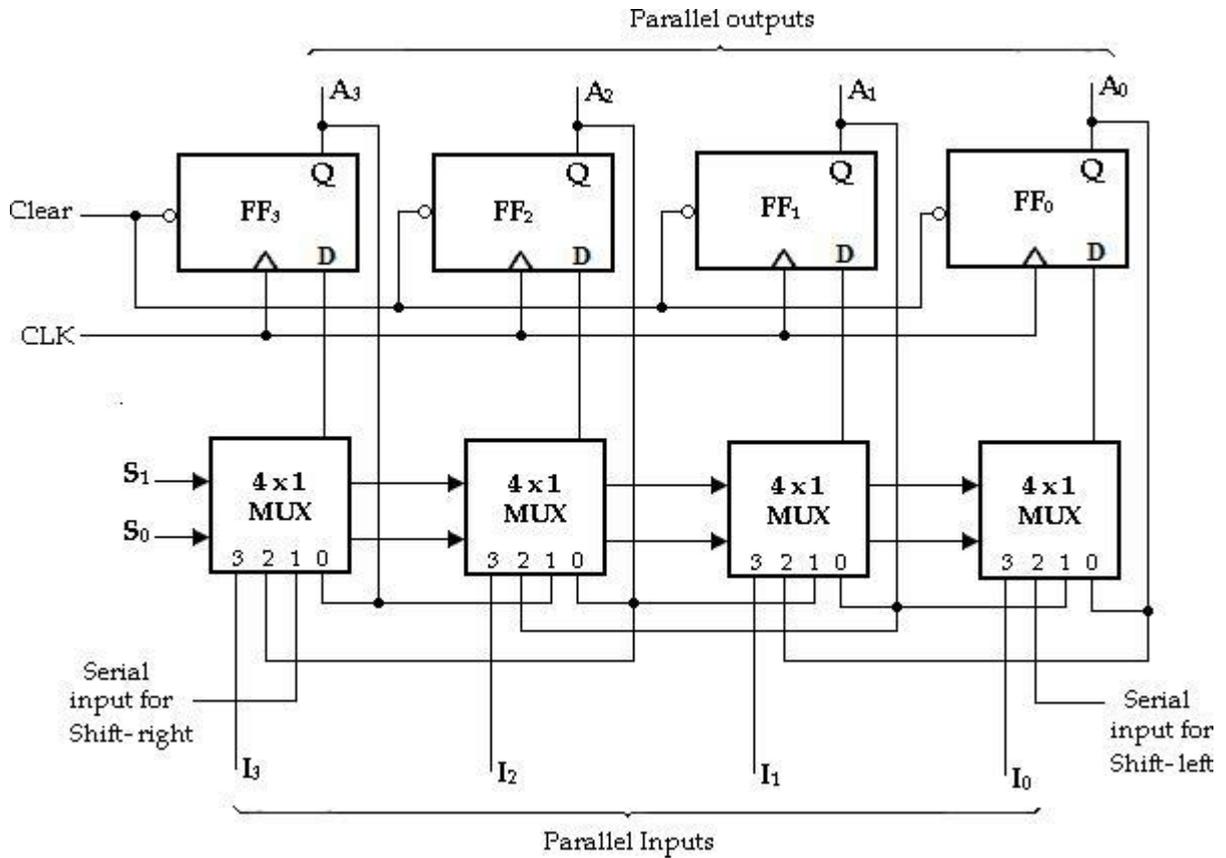
If the register has shift and parallel load capabilities, then it is called a shift register with parallel load or *universal shift register*. Shift register can be used for converting serial data to parallel data, and vice-versa. If a parallel load capability is added to a shift register, the data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

The functions of universal shift register are:

1. A clear control to clear the register to 0.
2. A clock input to synchronize the operations.
3. A shift-right control to enable the shift right operation and the serial input and output lines associated with the shift right.
4. A shift-left control to enable the shift left operation and the serial input and output lines associated with the shift left.
5. A parallel-load control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. ' n ' parallel output lines.
7. A control line that leaves the information in the register unchanged even though the clock pulses are continuously applied.

It consists of four D-Flip-Flops and four 4 input multiplexers (MUX). S_0 and S_1 are the two selection inputs connected to all the four multiplexers. These two selection inputs are used to select one of the four inputs of each multiplexer.

The input 0 in each MUX is selected when $S_1S_0 = 00$ and input 1 is selected when $S_1S_0 = 01$. Similarly inputs 2 and 3 are selected when $S_1S_0 = 10$ and $S_1S_0 = 11$ respectively. The inputs S_1 and S_0 control the mode of the operation of the register.



4-Bit Universal Shift Register

When $S_1S_0 = 00$, the present value of the register is applied to the D-inputs of the Flip-Flops. This is done by connecting the output of each Flip-Flop to the 0 input of the respective multiplexer. The next clock pulse transfers into each Flip-Flop, the binary value is held previously, and hence no change of state occurs.

When $S_1S_0 = 01$, terminal 1 of the multiplexer inputs has a path to the D inputs of the Flip-Flops. This causes a shift-right operation with the left serial input transferred into Flip-Flop FF₃.

When $S_1S_0 = 10$, a shift-left operation results with the right serial input going into Flip-Flop FF₁.

Finally when $S_1S_0 = 11$, the binary information on the parallel input lines (I_1 , I_2 , I_3 and I_4) are transferred into the register simultaneously during the next clock pulse.

The function table of bi-directional shift register with parallel inputs and parallel outputs is shown below.

Mode Control		Operation
S ₁	S ₀	
0	0	No change
0	1	Shift-right
1	0	Shift-left
1	1	Parallel load

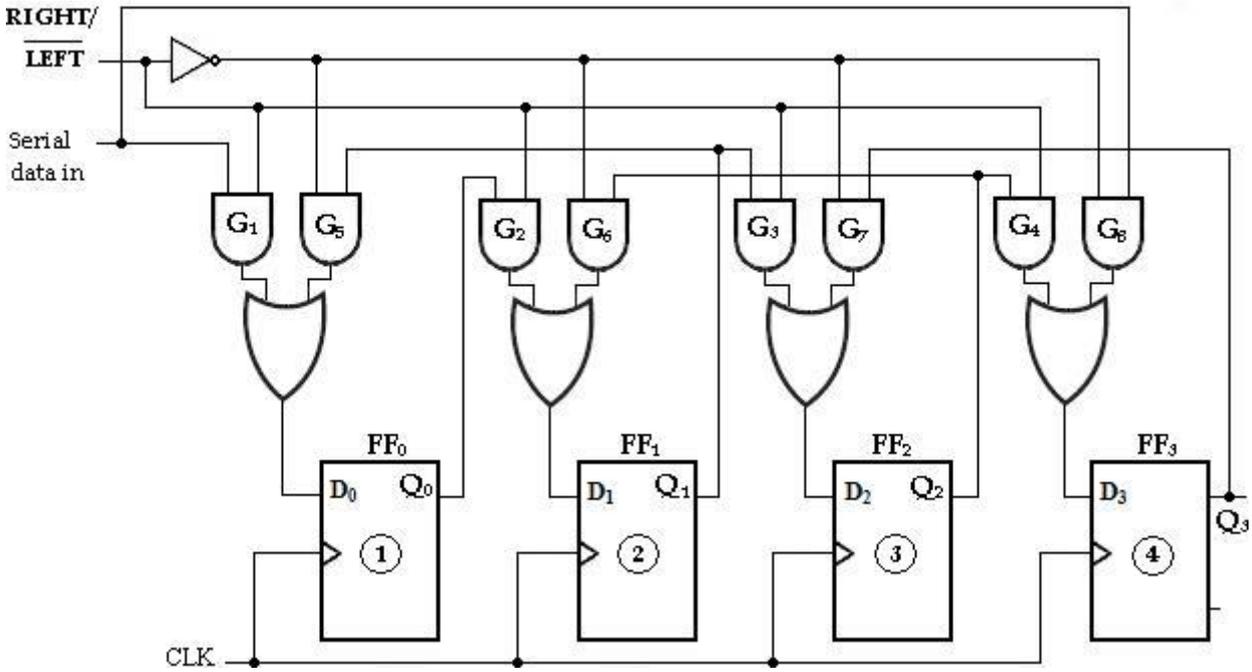
3.15.6 BI-DIRECTION SHIFT REGISTERS:

A bidirectional shift register is one in which the data can be shifted either left or right. It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left depending on the level of a control line.

A 4-bit bidirectional shift register is shown below. A HIGH on the RIGHT/LEFT control input allows data bits inside the register to be shifted to the right, and a LOW enables data bits inside the register to be shifted to the left.

When the RIGHT/LEFT control input is **HIGH**, gates G₁, G₂, G₃ and G₄ are enabled, and the state of the Q output of each Flip-Flop is passed through to the D input of the following Flip-Flop. When a clock pulse occurs, the data bits are shifted one place to the right.

When the RIGHT/LEFT control input is **LOW**, gates G₅, G₆, G₇ and G₈ are enabled, and the Q output of each Flip-Flop is passed through to the D input of the preceding Flip-Flop. When a clock pulse occurs, the data bits are then shifted one place to the left.



4-bit bi-directional shift register

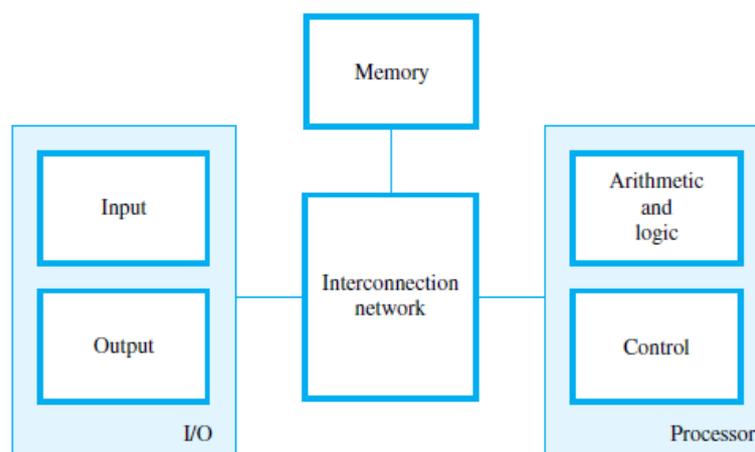
UNIT III COMPUTER FUNDAMENTALS

Functional Units of a Digital Computer: Von Neumann Architecture – Operation and Operands of Computer Hardware Instruction – Instruction Set Architecture (ISA): Memory Location, Address and Operation – Instruction and Instruction Sequencing – Addressing Modes, Encoding of Machine Instruction – Interaction between Assembly and High Level Language.

- Computer hardware consists of electronic circuits, magnetic and optical storage devices, displays, electromechanical devices, and communication facilities.
- Computer architecture encompasses the specification of an instruction set and the functional behaviour of the hardware units that implement the instructions.

FUNCTIONAL UNITS:

- A Computer is a fast electronic calculating machine that accepts digitized information from the user and processes it according to the sequence of instructions and provides the processed information to the user.
- A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output, and control units, as shown in Figure.
- The input unit accepts coded information from human operators using devices such as keyboards or from other computers over digital communication lines.
- The information received is stored in the computer's memory, either for later use or to be processed immediately by the arithmetic and logic unit. The processing steps are specified by a program that is also stored in the memory.
- Finally, the results are sent back to the outside world through the output unit.
- All of these actions are coordinated by the control unit. An interconnection network provides the means for the functional units to exchange information and coordinate their actions. We refer to the arithmetic and logic circuits, in conjunction with the main control circuits, as the processor. Input and output equipment is often collectively referred to as the input-output (I/O) unit.



Basic functional units of a computer.

- It is convenient to categorize this information as either instructions or data. Instructions, or machine instructions, are explicit commands that
 - Govern the transfer of information within a computer as well as between the computer and its I/O devices.
 - Specify the arithmetic and logic operations to be performed.

- The instructions and data handled by a computer must be encoded in a suitable format. Most present-day hardware employs digital circuits that have only two stable states.
- Each instruction, number, or character is encoded as a string of binary digits called *bits*, each having one of two possible values, 0 or 1, represented by the two stable states. Numbers are usually represented in positional binary notation. Alphanumeric characters are also expressed in terms of binary codes.

Input Unit

- Computers accept coded information through input units. The most common input device is the keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted to the processor.
- Many other kinds of input devices for human-computer interaction are available, including the touchpad, mouse, joystick, and trackball. These are often used as graphic input devices in conjunction with displays.
- Microphones can be used to capture audio input which is then sampled and converted into digital codes for storage and processing. Similarly, cameras can be used to capture video input.
- Digital communication facilities, such as the Internet, can also provide input to a computer from other computers and database servers.

Memory Unit

- The function of the memory unit is to store programs and data. There are two classes of storage, called primary and secondary.

Primary Memory

- Primary memory, also called main memory, is a fast memory that operates at electronic speeds. Programs must be stored in this memory while they are being executed. The memory consists of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are rarely read or written individually.
- Instead, they are handled in groups of fixed size called words. The memory is organized so that one word can be stored or retrieved in one basic operation. The number of bits in each word is referred to as the word length of the computer, typically 16, 32, or 64 bits.
- To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are consecutive numbers, starting from 0, that identify successive locations. A particular word is accessed by specifying its address and issuing a control command to the memory that starts the storage or retrieval process.
- Instructions and data can be written into or read from the memory under the control of the processor. It is essential to be able to access any word location in the memory as quickly as possible.
- A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called a random-access memory (RAM). The time required to access one word is called the memory access time. This time is independent of the location of the word

circuit chip. The purpose of the cache is to facilitate high instruction execution rates.

- At the start of program execution, the cache is empty. All program instructions and any required data are stored in the main memory. As execution proceeds, instructions are fetched into the processor chip, and a copy of each is placed in the cache. When the execution of an instruction requires data located in the main memory, the data are fetched and copies are also placed in the cache.
- Now, suppose a number of instructions are executed repeatedly as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. Similarly, if the same data locations are accessed repeatedly while copies of their contents are available in the cache, they can be fetched quickly.

Secondary Storage

- Although primary memory is essential, it tends to be expensive and does not retain information when power is turned off. Thus additional, less expensive, permanent secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.
- Access times for secondary storage are longer than for primary memory. A wide selection of secondary storage devices is available, including magnetic disks, optical disks (DVD and CD), and flash memory devices.

Arithmetic and Logic Unit

- Most computer operations are executed in the arithmetic and logic unit (ALU) of the processor. Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.
- For example, if two numbers located in the memory are to be added, they are brought into the processor, and the addition is carried out by the ALU. The sum may then be stored in the memory or retained in the processor for immediate use.
- When operands are brought into the processor, they are stored in high-speed storage elements called registers. Each register can store one word of data. Access times to registers are even shorter than access times to the cache unit on the processor chip.

Output Unit

- The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. A familiar example of such a device is a printer. Most printers employ either photocopying techniques, as in laser printers, or ink jet streams. Such printers may generate output at speeds of 20 or more pages per minute. However, printers are mechanical devices, and as such are quite slow compared to the electronic speed of a processor.
- Some units, such as graphic displays, provide both an output function, showing text and graphics, and an input function, through touch screen capability. The dual role of such units is the reason for using the single name input/output (I/O) unit in many cases.

- Control circuits are responsible for generating the timing signals that govern the transfers and determine when a given action is to take place.
- Data transfers between the processor and the memory are also managed by the control unit through timing signals. It is reasonable to think of a control unit as a well-defined, physically separate unit that interacts with other parts of the computer.
- Much of the control circuitry is physically distributed throughout the computer. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.

The operation of a computer can be summarized as follows:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

BASIC OPERATIONAL CONCEPTS:

- To perform a given task, an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be used as instruction operands are also stored in the memory.
- A typical instruction might be

Load R2, LOC

- This instruction reads the contents of a memory location whose address is represented symbolically by the label LOC and loads them into processor register R2.
- The original contents of location LOC are preserved, whereas those of register R2 are overwritten.
- Execution of this instruction requires several steps.
 - First, the instruction is fetched from the memory into the processor.
 - Next, the operation to be performed is determined by the control unit.
 - The operand at LOC is then fetched from the memory into the processor.
 - Finally, the operand is stored in register R2.

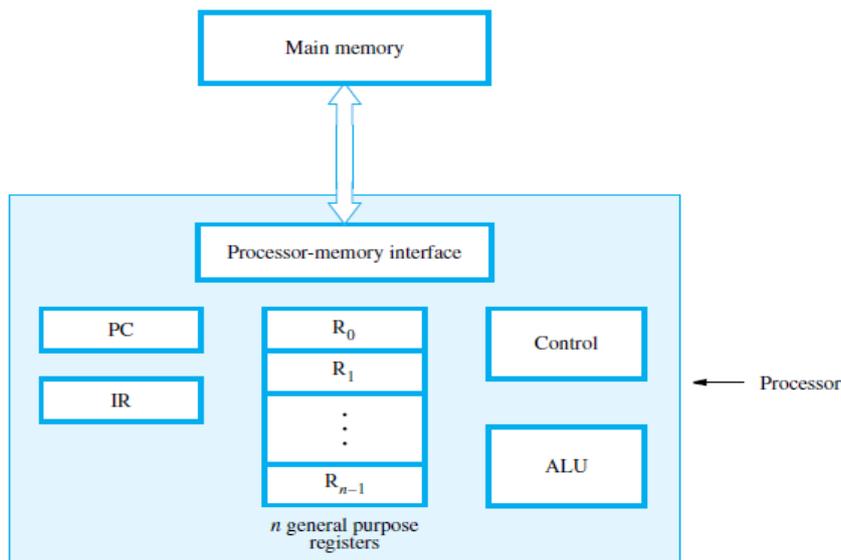
- After operands have been loaded from memory into processor registers, arithmetic or logic operations can be performed on them. For example, the instruction adds the contents of registers R2 and R3, then places their sum into register R4.

Add R4, R2, R3

- The operands in R2 and R3 are not altered, but the previous value in R4 is overwritten by the sum.

appropriate control signals. The data are then transferred to or from the memory.

- Figure shows how the memory and the processor can be connected. In addition to the ALU and the control circuitry, the processor contains a number of registers used for several different purposes.
- The instruction register (IR) holds the instruction that is currently being executed. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.
- The program counter (PC) is another specialized register. It contains the memory address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed.
- In addition to the IR and PC, Figure shows general-purpose registers R_0 through R_{n-1} , often called processor registers.



Connection between the processor and the main memory.

- The processor-memory interface is a circuit which manages the transfer of data between the main memory and the processor. If a word is to be read from the memory, the interface sends the address of that word to the memory along with a Read control signal.
- The interface waits for the word to be retrieved, then transfers it to the appropriate processor register.
- If a word is to be written into memory, the interface transfers both the address and the word to the memory along with a Write control signal.
- **Let us now consider some typical operating steps.** A program must be in the main memory in order for it to be executed. It is often transferred there from secondary storage through the input unit.
- Execution of the program begins when the PC is set to point to the first instruction of the program. The contents of the PC are transferred to the memory along with a Read control signal.

- When the addressed word (in this case, the first instruction of the program) has been fetched from the memory it is loaded into register IR. At this point, the instruction is ready to be interpreted and executed.
- Instructions such as Load, Store, and Add perform data transfer and arithmetic operations. If an operand that resides in the memory is required for an instruction, it is fetched by sending its address to the memory and initiating a Read operation.
- When the operand has been fetched from the memory, it is transferred to a processor register. After operands have been fetched in this way, the ALU can perform a desired arithmetic operation, such as Add, on the values in processor registers. The result is sent to a processor register.
- If the result is to be written into the memory with a Store instruction, it is transferred from the processor register to the memory, along with the address of the location where the result is to be stored, then a Write operation is initiated.
- At some point during the execution of each instruction, the contents of the PC are incremented so that the PC points to the next instruction to be executed. Thus, as soon as the execution of the current instruction is completed, the processor is ready to fetch a new instruction.
- In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions are provided for the purpose of handling I/O transfers.
- Normal execution of a program may be preempted if some device requires urgent service. In order to respond immediately, execution of the current program must be suspended.
- To cause this, the device raises an interrupt signal, which is a request for service by the processor. The processor provides the requested service by executing a program called an interrupt-service routine.
- Normally, the information that is saved includes the contents of the PC, the contents of the general-purpose registers, and some control information. When the interrupt-service routine is completed, the state of the processor is restored from the memory so that the interrupted program may continue.

PERFORMANCE:

- Computer performance is based on response time and throughput (bandwidth). When trying to choose among different computers, performance is an important attribute. Accurately measuring and comparing different computers is critical to purchasers and therefore to designers.
- As an individual computer user, we are interested in reducing response time. **Response time** is the time between the start and completion of a task and also called as execution time.
- The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time and so on.
- In general, data centre managers are interested in increasing throughput. Throughput is also called as bandwidth is another measure of performance; it is the number of tasks completed per unit area. It is the amount of work done in a given time.

Performance evaluation

- To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_x = \frac{1}{\text{Execution time}_x}$$

- This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\frac{\text{Performance}_X}{\text{Execution time}_X} > \frac{\text{Performance}_Y}{\text{Execution time}_Y}$$

$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

- For two processors X and Y, relative performance (n) is defined as,

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

- If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Measuring Performance

- Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program execution time is measured in seconds per program.
- **CPU execution time** or simply **CPU time**, is the time the CPU spends computing for specific task and does not include time spent waiting for I/O or running other programs.
- CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**.

CPU Performance and Its Factors

- A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU Execution time} = \text{CPU clock cycles for a program} \times \text{clock cycle time}$$

- Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU Execution Time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Instruction Performance

- The performance equations above did not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program.
- Execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

- The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**.

The Classic CPU Performance Equation

- The basic performance equation in terms of **instruction count** (the number of instructions executed by the program), CPI, and clock cycle time:

$$CPU\ Time = Instructions\ count \times CPI \times Clock\ cycle\ time$$

- or, since the clock rate is the inverse of clock cycle time:

$$CPU\ Time = \frac{Instruction\ count \times CPI}{Clock\ rate}$$

INSTRUCTIONS:

- The words of a computer's language are called instructions, and its vocabulary is called an instruction set. The idea that instructions and data of many type can be stored in memory is known as stored program concept.

OPERATIONS OF THE COMPUTER HARDWARE:

- Every computer must be able to perform arithmetic. The MIPS assembly language notation

add a, b, c

instructs a computer to add the two variables b and c and to put their sum in a.

- Each MIPS arithmetic instruction performs only one operation and must always have exactly three variables. The following sequence of instructions adds the four variables:

add a, b, c # The sum of b and c is placed in a

add a, a, d # The sum of b, c, and d is now in a

add a, a, e # The sum of b, c, d, and e is now in a

- Thus, it takes three instructions to sum the four variables. The words to the right of the sharp symbol (#) on each line above are comments for the human reader, so the computer ignores them.

MIPS operands

Name	Example	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

- Figure shows that MIPS use either registers or memory locations as operands. MIPS uses 32 registers for storing data temporarily and for fast access. Memory locations are accessed by data transfer instructions.
- Figure shows that MIPS assembly language. There are five categories of instructions namely, arithmetic, data transfer, logical, conditional branch and unconditional jump.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	Three register operands
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	Three register operands
	add immediate	addi \$s1,\$s2,20	$\$s1 = \$s2 + 20$	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Word from memory to register
	store word	sw \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Word from register to memory
	load half	lh \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Halfword memory to register
	store half	sh \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1$	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	$\$s1 = \text{Memory}[\$s2 + 20]$	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	$\text{Memory}[\$s2 + 20] = \$s1; \$s1 = 0 \text{ or } 1$	Store word as 2nd half of atomic swap
	load upper immed.	lui \$s1,20	$\$s1 = 20 * 2^{16}$	Loads constant in upper 16 bits
Logical	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \$s3$	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim (\$s2 \$s3)$	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	$\$s1 = \$s2 \& 20$	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	$\$s1 = \$s2 20$	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	If $(\$s1 == \$s2)$ go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	If $(\$s1 \neq \$s2)$ go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	If $(\$s2 < \$s3)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	If $(\$s2 < \$s3)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	If $(\$s2 < 20)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	If $(\$s2 < 20)$ $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant unsigned
Unconditional jump	Jump	j 2500	go to 10000	Jump to target address
	Jump register	jr \$ra	go to \$ra	For switch, procedure return
	Jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	For procedure call

OPERANDS OF THE COMPUTER HARDWARE:

- In MIPS instruction set architecture, operand can either in register or memory. Most of the arithmetic and logical instructions use register operands.
- Three types of operands are used in MIPS: register operands, memory operands and constant or immediate operands.

Register operands

- Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers.
- The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the MIPS architecture.

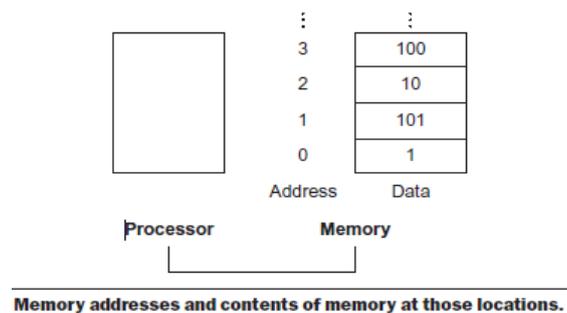
- We will use \$s0, \$s1, . . . for registers that correspond to variables in C and Java programs and \$t0, \$t1, . . . for temporary registers needed to compile the program into MIPS instructions.

add \$t0,\$s1,\$s2

- Data is more useful when in register. A MIPS arithmetic instruction can read two registers, operate on them, and write the result. A MIPS data transfer instruction only reads one operand or writes one operand, without operating on it. Thus, registers take less time to access and have higher throughput than memory.

Memory operands

- The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.
- Arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**.
- To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure, the address of the third data element is 2, and the value of Memory [2] is 10.



- The data transfer instruction that copies data from memory to a register is traditionally called load. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The actual MIPS name for this instruction is lw, standing for load word.

lw \$t0,8(\$s3)

- Computers divide into those that use the address of the left most or “**big end**” byte as the word address versus those that use the rightmost or “**little end**” byte. MIPS is in the big-endian camp.
- The instruction complementary to load is traditionally called store; it copies data from a register to memory. The actual MIPS name is sw, standing for store word.
- The process of putting less commonly used variables (or those needed later) into memory is called spilling registers.

Constant or Immediate Operands

- Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array.
- For example, to add the constant 4 to register \$s3, we could use the code

addi \$s3,\$s3,4

\$s3 = \$s3 + 4

- Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

INSTRUCTION REPRESENTATION:

- Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.
- In MIPS assembly language, registers \$s0 to \$s7 map onto registers 16 to 23, and registers \$t0 to \$t7 map onto registers 8 to 15.
- Hence, \$s0 means register 16, \$s1 means register 17, \$s2 means register 18, . . . , \$t0 means register 8, \$t1 means register 9, and so on.
- Two types of instruction formats are used in MIPS. They are
 - R-type (register) or R-format
 - I-type (immediate) or I-format

R-type

- MIPS fields are given names to make them easier to discuss:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Here is the meaning of each name of the fields in MIPS instructions:
 - op: Basic operation of the instruction, traditionally called the **opcode**.
 - rs: The first register source operand.
 - rt: The second register source operand.
 - rd: The register destination operand. It gets the result of the operation.
 - shamt: Shift amount.
 - funct: Function. This field, oft en called the function code, selects the specific variant of the operation in the op field.

add \$t0,\$s1,\$s2 # Temporary reg \$t0=\$s2+\$s1

0	17	18	8	0	32
---	----	----	---	---	----

I-type

- The field of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- The 16-bit address means a load word instruction can load any word with in a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than 2^{15} .

lw \$t0,32(\$s3)

Temporary reg \$t0 gets A[8]

- Here, 19 (for \$s3) is placed in the rs field,
- 8 (for \$t0) is placed in the rt field, and
- 32 is placed in the address field.
- 35 is placed in op field for lw instruction.
- The meaning of the rt field has changed for this instruction: in a load word instruction, the rt field specifies the destination register, which receives the result of the load.

LOGICAL OPERATIONS:

- Logical operations work on bits. The following is the list of logical operations supported in MIPS:

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit NOT	~	~	nor

C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

Sll and srl

- The first class of such operations is called shift s. They move all the bits in a word to the left or right, filling the emptied bits with 0s. The actual name of the two MIPS shift instructions are called shift left logical (sll) and shift right logical (srl).

sll \$t2,\$s0,4

reg \$t2 = reg \$s0 << 4 bits

- The shamt field in the R-format is used in shift instructions, it stands for shift amount. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

- The encoding of sll is 0 in both the op and funct fields, rd contains 10 (register \$t2), rt contains 16 (register \$s0), and shamt contains 4. The rs field is unused and thus is set to 0.

and

- AND is a bit by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register \$t2 contains

- For example, if register \$t2 contains

0000 0000 0000 0000 0000 1101 1100 0000_{two}

and register \$t1 contains

0000 0000 0000 0000 0011 1100 0000 0000_{two}

then, after executing the MIPS instruction

and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

the value of register \$t0 would be

0000 0000 0000 0000 0000 1100 0000 0000_{two}

- AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a mask, since the mask “conceals” some bits.

or

- It is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1. To elaborate, if the registers \$t1 and \$t2 are unchanged from the preceding example, the result of the MIPS instruction

or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

is this value in register \$t0:

0000 0000 0000 0000 0011 1101 1100 0000_{two}

not

- A logical bit-by bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

nor

- A logical bit-by bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in both operands.
- If the register \$t1 is unchanged from the preceding example and register \$t3 has the value 0, the result of the MIPS instruction

nor \$t0,\$t1,\$t3 # reg \$t0 = ~(reg \$t1 | reg \$t3)

is this value in register \$t0:

1111 1111 1111 1111 1100 0011 1111 1111_{two}

- Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions and immediate (andi) and or immediate (ori). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

DECISION MAKING:

- Decision making is commonly represented in programming languages using the if statement, sometimes combined with go to statements and labels. MIPS assembly language includes two decision-making instructions, similar to an if statement with a go to. The first instruction is

beq register1, register2, L1

- This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal. The second instruction is

bne register1, register2, L1

- It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. The mnemonic bne stands for branch if not equal. These two instructions are traditionally called **conditional branches**.

Loops

- Decisions are important both for choosing between two alternatives—found in if statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.
- In MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called **set on less than, or slt**. For example,

slt \$t0, \$s3, \$s4 # \$t0 = 1 if \$s3 < \$s4
in register \$s4; otherwise, register \$t0 is set to 0.

- Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register \$s2 is less than the constant 10, we can just write

slti \$t0,\$s2,10 # \$t0 = 1 if \$s2 < 10

Case/Switch Statement

- Most programming languages have a case or switch statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement switch is via a sequence of conditional tests, turning the switch statement into a chain of if-then-else statements.
- Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence.
- The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register.
- To support such situations, computers like MIPS include a jump register instruction (jr), meaning an unconditional jump to the address specified in a register.
- In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register \$ra. The **jump-and-link instruction** (jal) is simply written

jal ProcedureAddress

- The link portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This “link,” stored in register \$ra (register 31), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.
- To support such situations, computers like MIPS use jump register instruction (jr), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:

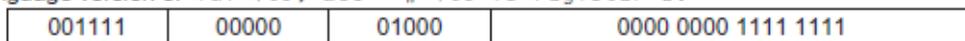
jr \$ra

MIPS ADDRESSING:

32-Bit Immediate Operands

- The MIPS instruction set includes the instruction load upper immediate (lui) specifically to set the upper 16 bits of a constant in a register, allowing a subsequent instruction to specify the lower 16 bits of the constant. Figure 2.17 shows the operation of lui.

The machine language version of `lui $t0, 255 # $t0 is register 8:`



Contents of register \$t0 after executing `lui $t0, 255:`



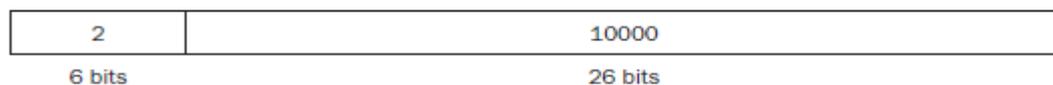
The effect of the lui instruction. The instruction `lui` transfers the 16-bit immediate constant field value into the leftmost 16 bits of the register, filling the lower 16 bits with 0s.

Addressing in Branches and Jumps

- The MIPS jump instructions have the simplest addressing. They use the final MIPS instruction format, called the J-type, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

`j 10000 # go to location 10000`

could be assembled into this format

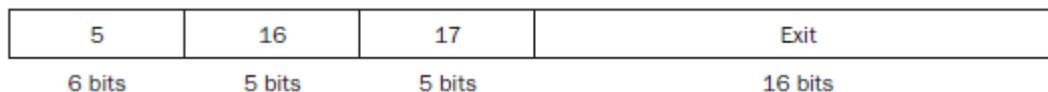


where the value of the jump opcode is 2 and the jump address is 10000.

- Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address. Thus,

`bne $s0,$s1,Exit # go to Exit if $s0 ≠ $s1`

is assembled into this instruction, leaving only 16 bits for the branch address:



MIPS addressing modes:

- Multiple forms of addressing are generically called addressing modes. The MIPS addressing modes are the following:

1. Immediate addressing:

- Immediate addressing, where the operand is a constant within the instruction itself.

1. Immediate addressing

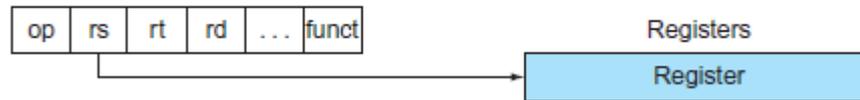


- **Example:** `lui $s0,61` #decimal 61 is loaded into upper 16 bits of \$s0.

2. Register addressing:

- Register addressing, where the operand is a register

2. Register addressing

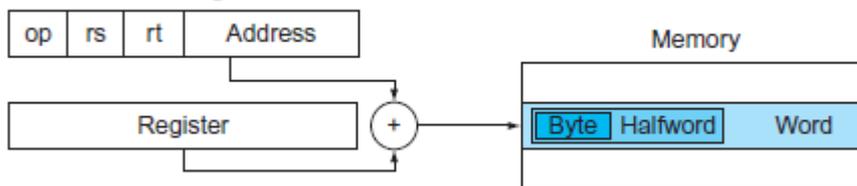


- **Example:** `add $t1,$s1,$s2` # $\$t1 = \$s1 + \$s2$. In the above instruction, \$s1, \$s2 are source registers rs and rt, \$t1 is the destination register rd.

3. Base or Displacement addressing:

- Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction.

3. Base addressing

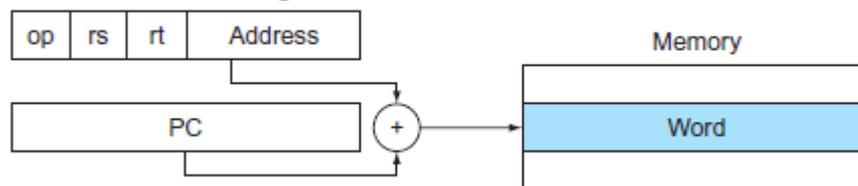


- **Example:** `lw $t0,32($t1)` # Temp reg \$t0=save[i]. In the above example, \$t1 is the base register and the operand is available in memory whose memory address is sum of base register \$t1 and offset 32.

4. PC relative addressing:

- PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction.

4. PC-relative addressing

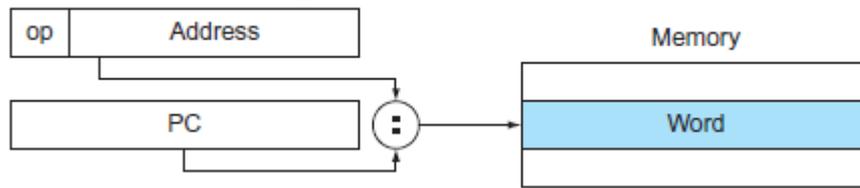


- **Example:** `bne $s0,$s1,Exit` #go to Exit if \$s0 \neq \$s1. In the above example branch address is calculated by adding the PC value with the constant in the instruction.

5. Pseudo-direct addressing:

- Pseudo-direct addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

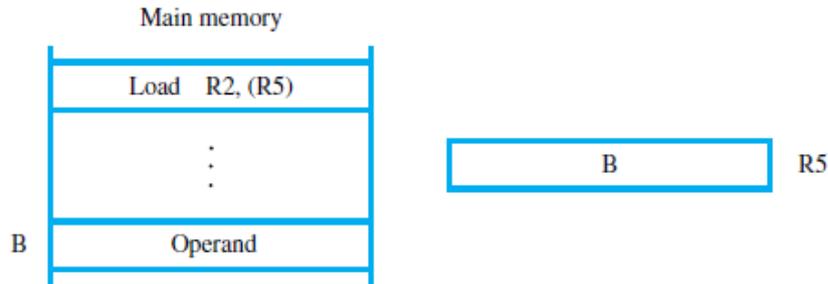
5. Pseudodirect addressing



➤ **Example:** j 2500 # go to location 10000₁₀

6. Indirect mode:

➤ The effective address of the operand is the contents of a register that is specified in the instruction.

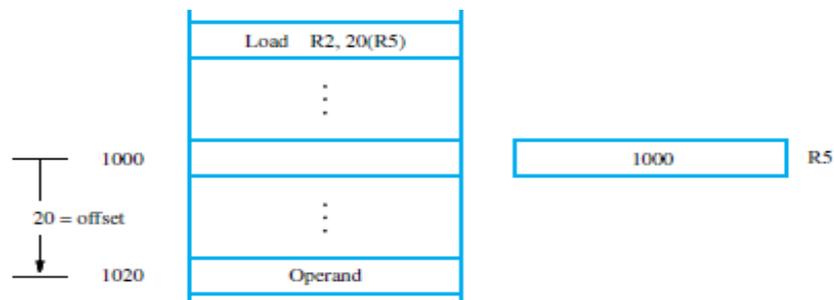


Register indirect addressing.

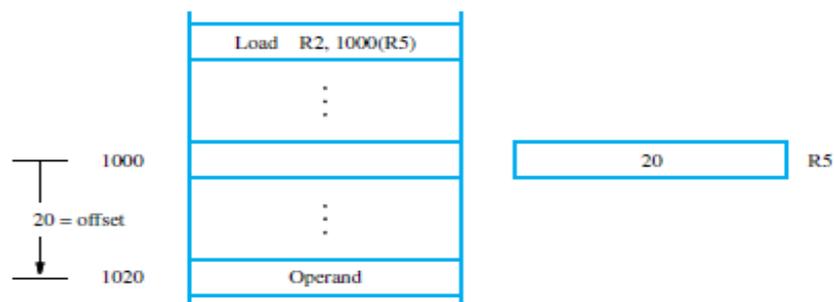
➤ To execute the Load instruction in Figure 2.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory. The value from the memory is the desired operand, which the processor loads into register R2.

7. Index mode:

➤ The effective address of the operand is generated by adding a constant value to the contents of a register.



(a) Offset is given as a constant



(b) Offset is in the index register

Indexed addressing.

➤ In Figure a, the index register, R5, contains the address of a memory location, and the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found.

- An alternative use is illustrated in Figure *b*. Here, the constant *X* corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is held in a register.

8. Auto increment/ decrement mode:

- The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented / decremented to point to the next item in a list.

UNIT IV PROCESSOR

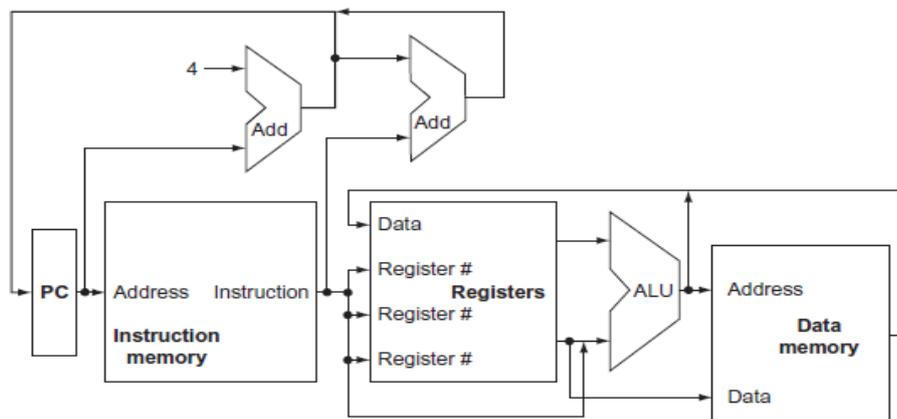
Instruction Execution – Building a Data Path – Designing a Control Unit – Hardwired Control, Microprogrammed Control – Pipelining – Data Hazard – Control Hazards.

A BASIC MIPS IMPLEMENTATION:

- We will be examining an implementation that includes a subset of the core MIPS instruction set:
 - The memory-reference instructions load word (lw) and store word (sw)
 - The arithmetic-logical instructions add, sub, AND, OR, and slt
 - The instructions branch equal (beq) and jump (j), which we add last

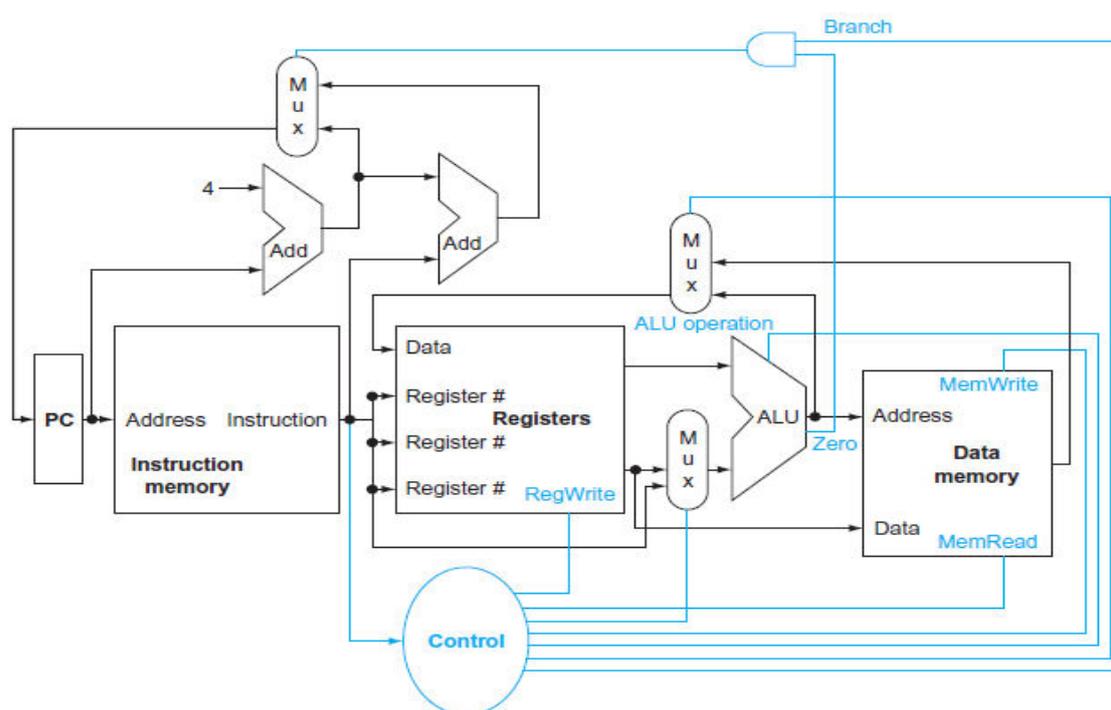
An Overview of the Implementation

- For every instruction, the first two steps are identical:
 1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
 2. Read one or two registers, using fields of the instruction to select the registers to read.
 3. After these two steps, the actions required to complete the instruction depend on the instruction class. all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers.
 - The memory-reference instructions use the ALU for an address calculation,
 - the arithmetic-logical instructions for the operation execution, and
 - branches for comparison.
 4. After using the ALU, the actions required to complete various instruction classes differ.
 - A memory-reference instruction will need to access the memory either to read data for a load or write data for a store.
 - An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register.
 - Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by 4 to get the address of the next instruction.
 5. Finally, the result from ALU is stored back in memory or to a register.
- Figure shows the high-level view of a MIPS implementation, focusing on the various functional units and their interconnection.



An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

- All instructions start by using the program counter to supply the instruction address to the instruction memory.
- After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or a compare (for a branch).
- If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers.
- The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.
- Figure shows data going to a particular unit as coming from two different sources. For example,
 - The value written into the PC can come from one of two adders,
 - The data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a register or the immediate field of the instruction.
 - The selection is commonly done with a device called a multiplexor, although this device might better be called a data selector.



The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

- A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.
- The top multiplexor (“Mux”) controls what value replaces the PC (PC + 4 or the branch destination address)
- The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.

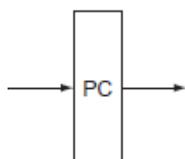
- Finally, the bottommost multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store).

BUILDING A DATAPATH:

- A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

1. program counter:

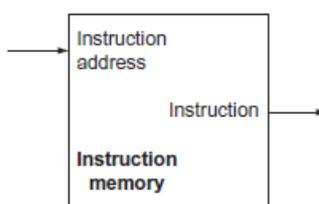
- Program counter (PC) is a 32-bit register. It holds the address of the current instruction. It does not need a write control signal.



b. Program counter

2. Instruction memory:

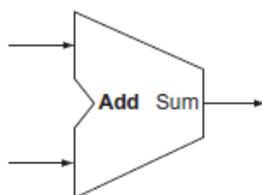
- Instruction memory is a memory unit that is used to store the instructions of a program. It get address as input and outputs the instruction which is stored in the given address.
- The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, no read control signal is needed.



a. Instruction memory

3. Adder:

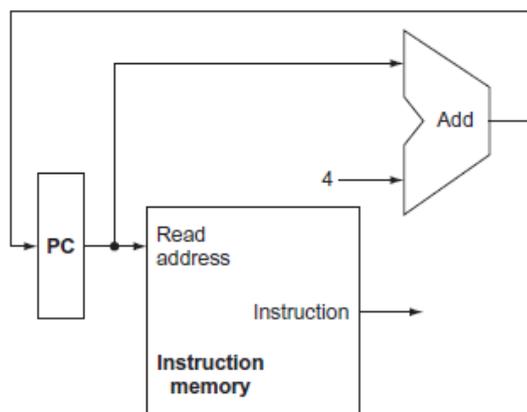
- The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output. The adder is also used to increment the PC to the address of the next instruction. It is combinational circuit that can built from the ALU.



c. Adder

- To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later.

- Figure shows how to combine the three elements from above Figure to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction. The fetched instruction is used by other parts of the datapath.



A portion of the datapath used for fetching instructions and incrementing the program counter.

4. Register file:

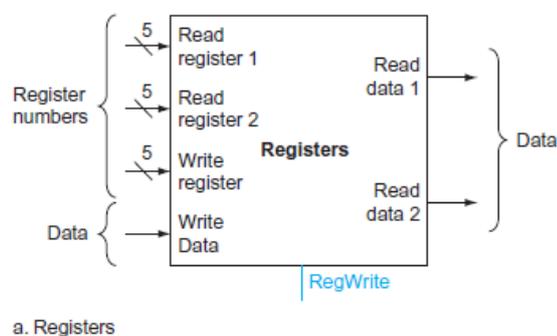
- The processor's 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file.
- The register file contains the register state of the computer. In addition, we will need an ALU to operate on the values read from the registers.

Read

- For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. No other control lines are needed.

Write

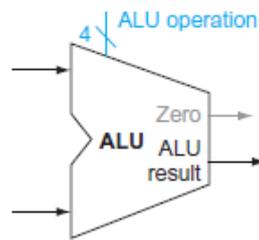
To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register. The write operation needs a write control signal.



- We need a total of four inputs (three for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$), whereas the data input and two data output buses are each 32 bits wide.

5. ALU:

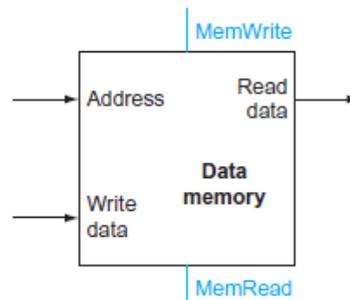
- ALU, which takes two 32-bit inputs and produces a 32-bit result, as well as a 1-bit signal if the result is 0. The operation to be performed by the
- ALU is controlled with the ALU operation signal, which will be 4 bits wide. We will use the Zero detection output of the ALU shortly to implement branches.



b. ALU

6. Data memory:

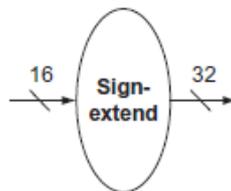
- The memory unit is a state element with two inputs, an address and a write data and a single output called read data. There are separate read (MemRead) and write control signals (MemWrite)



a. Data memory unit

7. Sign extended:

- It is used to increase the size of a data item by replicating the high-order sign bit of the original data item in the high order bits of the larger, destination data item.
- It is used to sign-extend the 16-bit offset field in the instruction to a 32-bit signed value.



b. Sign extension unit

Data path elements for branches:

- In MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branches.
- The address specified in a branch, which becomes the new program counter (PC) if the branch is taken.
- The data path for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits.

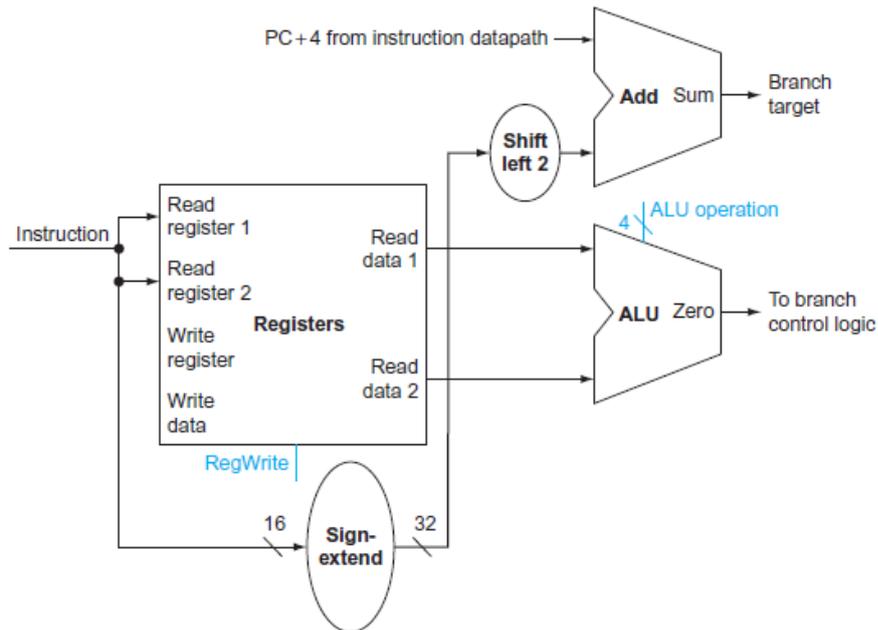
- Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

Branch taken

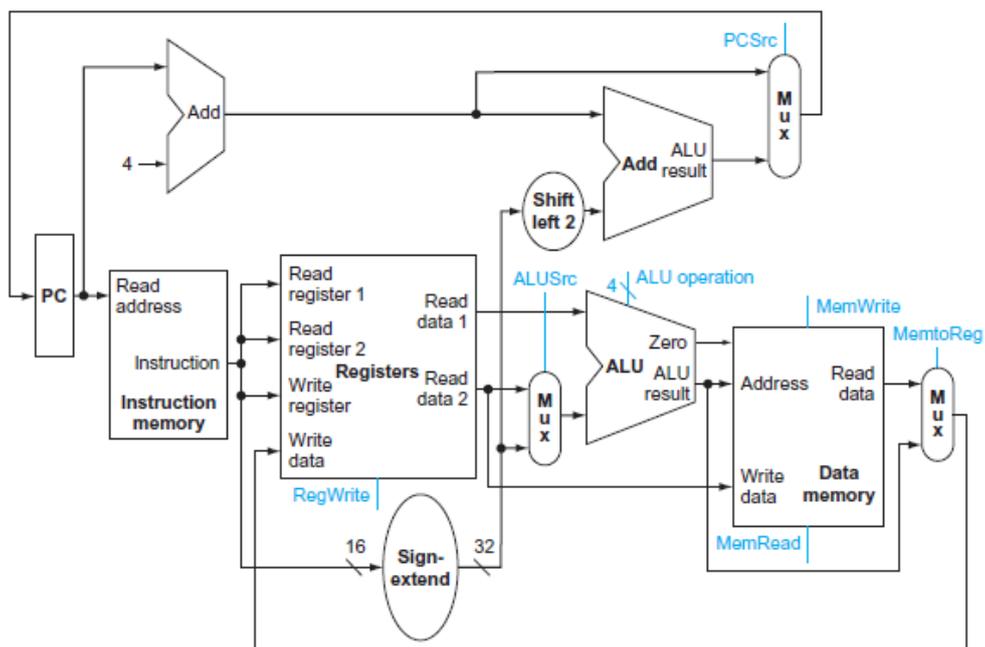
- A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.

Branch not taken or (untaken branch)

- A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.



Data path elements for branches



The simple datapath for the core MIPS architecture combines the elements required by different instruction classes.

CONTROL IMPLEMENTATION SCHEME:

- This simple implementation covers load word (lw), store word (sw), branch equal (beq), and the arithmetic-logical instructions add, sub, AND, OR, and set on less than. We will later enhance the design to include a jump instruction (j).

ALU control:

- The MIPS ALU defines the 6 following combinations of four control inputs:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

- Depending on the instruction class, the ALU will need to perform one of these first five functions.
- For load word and store word instructions, ALU is used to compute the memory address by addition.
- For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field.
- For branch equal, the ALU must perform a subtraction.
- We can generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field, called ALUOp. ALUOp indicates whether the operation to be performed:
 - 00- add for loads and stores,
 - 01- subtract for beq,
 - 10- determined by the operation encoded in the funct field.
 - The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations.
- The following table shows the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. The opcode, listed in the first column, determines the setting of the ALUOp bits.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

- ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is

shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

Designing the Main Control Unit

- To understand how to connect the fields of an instruction to the data path, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions. Figure shows these formats.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

R-type instruction:

- Instruction format for R-format instructions, which all have an opcode of 0.
- These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination.
- The ALU function is in the funct field and is decoded by the ALU control design in the previous section.
- The R-type instructions that we implement are add, sub, AND, OR, and slt.
- The shamt field is used only for shifts.

Load and store instruction:

- Instruction format for load (opcode = 35_{10}) and store (opcode = 43_{10}) instructions.
- The register rs is the base register that is added to the 16-bit address field to form the memory address.
- For loads, rt is the destination register for the loaded value.
- For stores, rt is the source register whose value should be stored into memory.

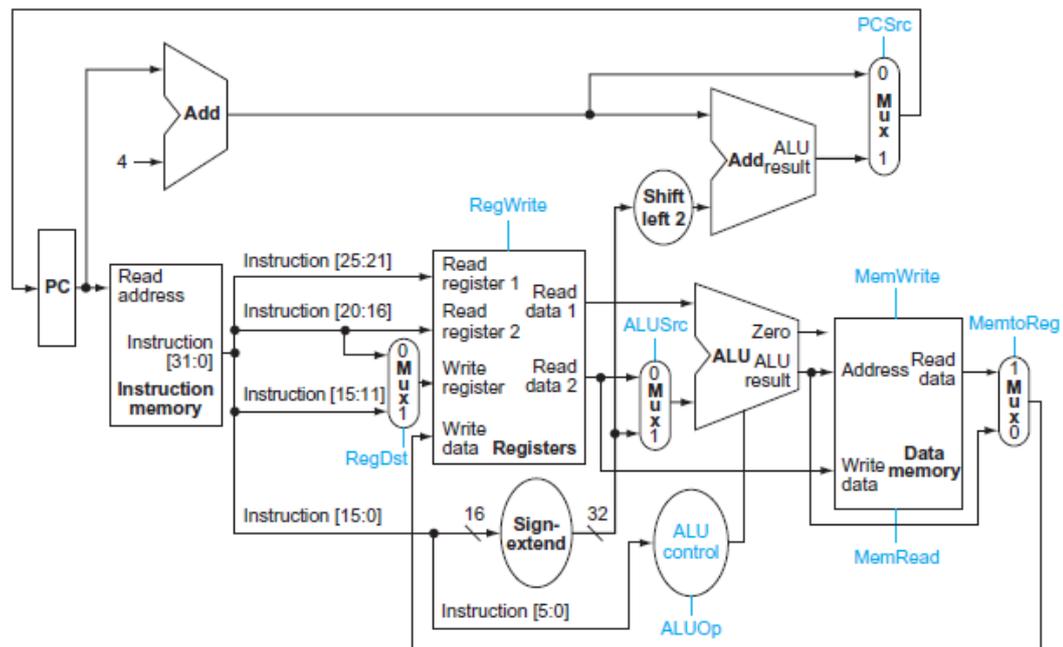
Branch instruction:

- Instruction format for branch equal (opcode = 4).
- The registers rs and rt are the source registers that are compared for equality.
- The 16-bit address field is sign-extended, shifted, and added to the PC + 4 to compute the branch target address.

There are several major observations about this instruction format that we will rely on:

- The op field is always contained in bits 31:26.
- The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and store.
- The base register for load and store instructions is always in bit positions 25:21 (rs).

- The 16-bit off set for branch equal, load, and store is always in positions 15:0.
- The destination register is in one of two places. For a load it is in bit positions 20:16 (rt), while for an R-type instruction it is in bit positions 15:11 (rd). Thus, we will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

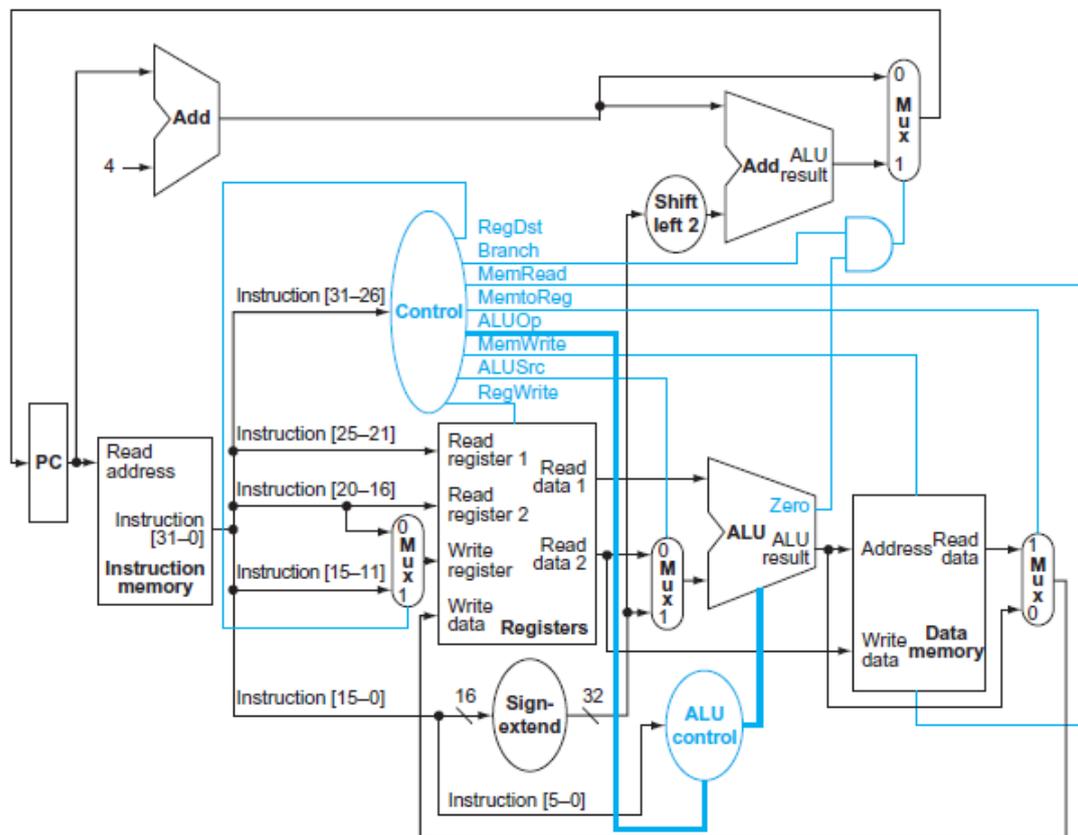


Simple data path with multiplexors and control lines

Control signals:

- The following table shows the effect of each of the seven control signals when asserted and deasserted.

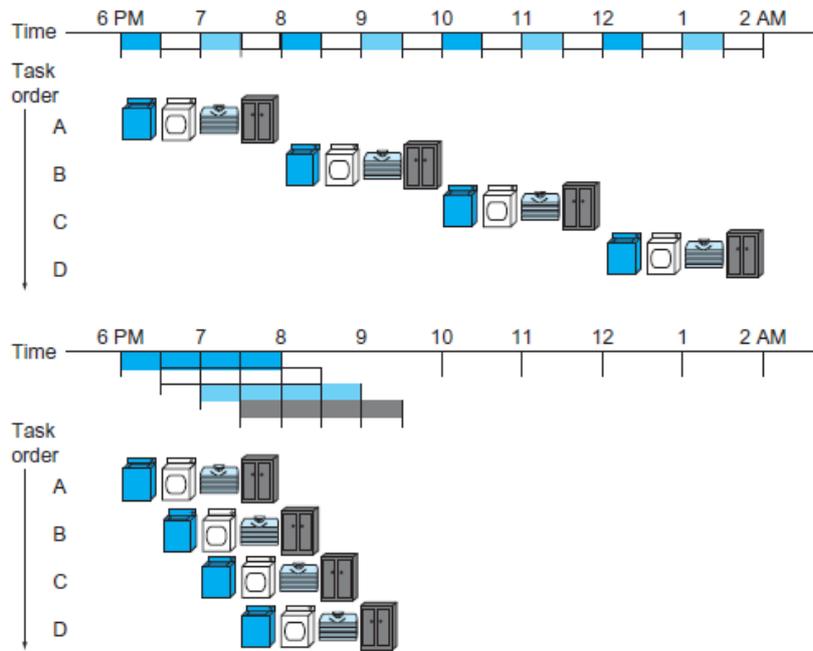
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.



The simple data path with control unit

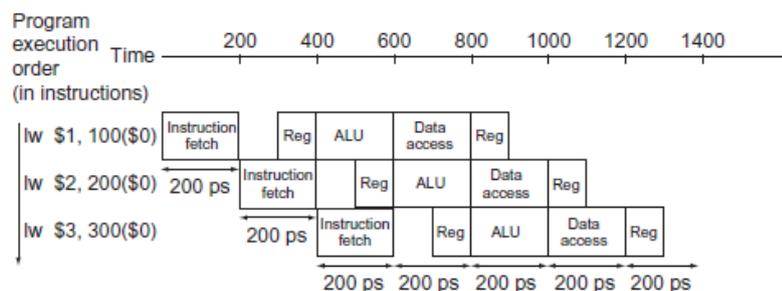
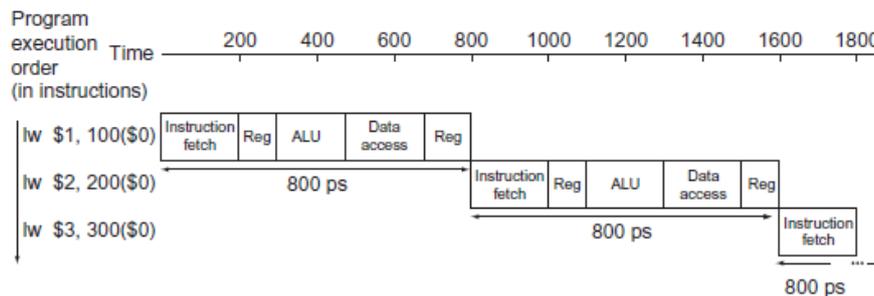
PIPELINING:

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution. It exploits parallelism among the instructions in a sequential instruction stream. It has an important advantage that is basically invisible to the programmer.
- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction.
- It improves the instruction throughput rather than individual instruction execution time or latency.
- The concept of pipelining can be understood by Laundry analogy. The non-pipelined approach to laundry would be as follows:
 - Place one dirty load of clothes in the washer.
 - When the washer is finished, place the wet load in the dryer
 - When the dryer is finished, place the dry load on a table and fold.
 - When folding is finished, ask your roommate to put the clothes away.
- When your roommate is done, start over with the next dirty load.
- The pipelined approach takes much less time, as Figure shows. As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load.
- When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer.
- Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer.
- At this point all steps—called stages in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks.



The laundry analogy for pipelining.

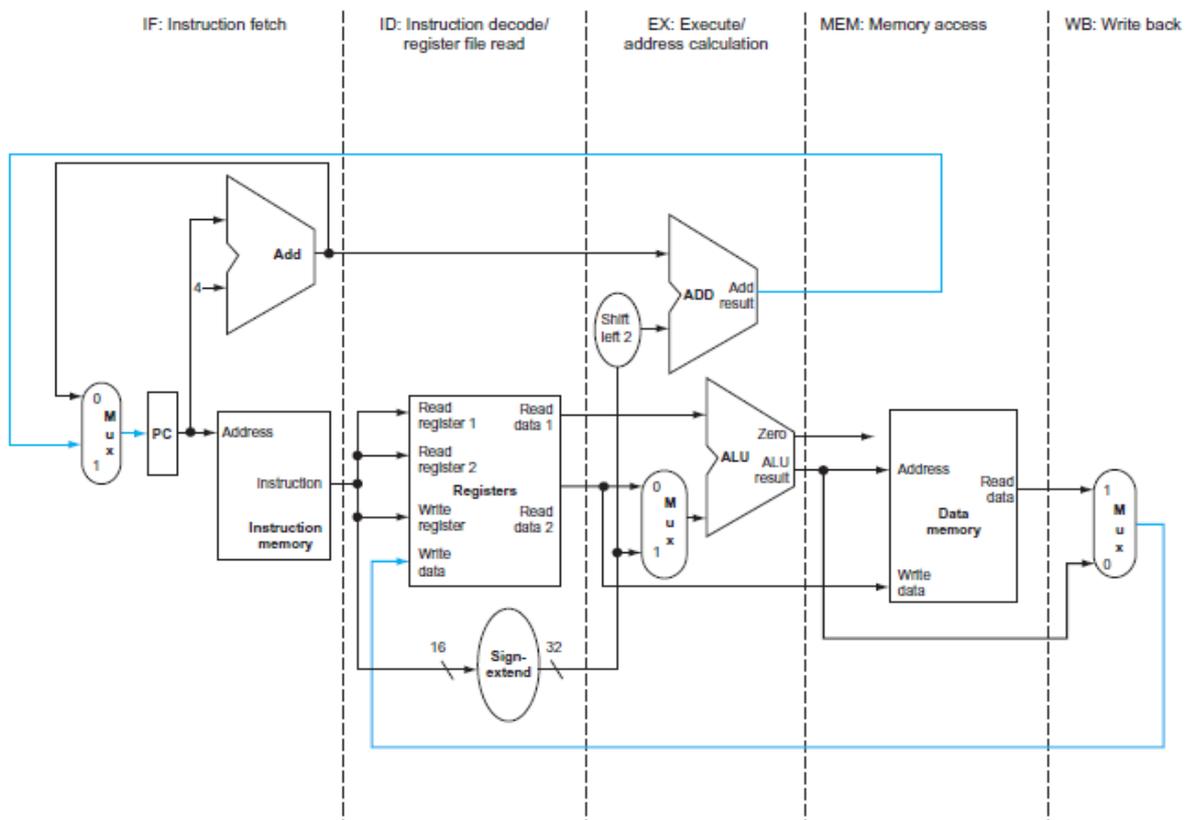
- Pipelining improves throughput of our laundry system. Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.
- The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours.
- The same principles apply to processors where we pipeline instruction-execution. MIPS instructions classically take five steps:
 1. Fetch instruction from memory.
 2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
 3. Execute the operation or calculate an address.
 4. Access an operand in data memory.
 5. Write the result into a register.



Single-cycle, nonpipelined execution in top versus pipelined execution in bottom.

PIPELINED DATA PATH AND CONTROL

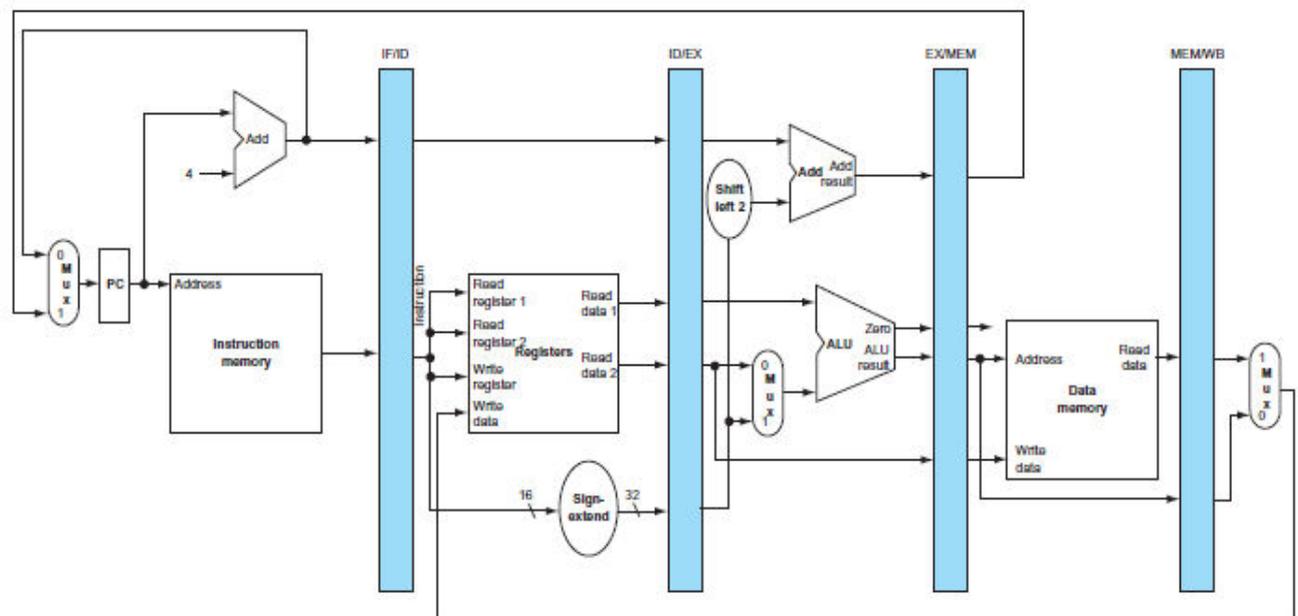
- Figure shows the single-cycle data path with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the data path into five pieces, with each piece named corresponding to a stage of instruction execution:
 1. IF: Instruction fetch
 2. ID: Instruction decode and register file read
 3. EX: Execution or address calculation
 4. MEM: Data memory access
 5. WB: Write back
- Instructions and data move generally from left to right through the five stages as they complete execution. There are, however, two exceptions to this left-to-right flow of instructions:
 - The write-back stage, which places the result back into the register file in the middle of the data path
 - The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.



Single cycle data path with pipeline stages

Pipelined data path:

- Figure 4.35 shows the pipelined data path with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID. There is no pipeline register at the end of the write-back stage.



Pipelined data path

- The pipeline registers, separate each pipeline stage. They are labelled by the stages that they separate; for example, the first is labelled IF/ID because it separates the instruction fetch and instructions decode stages.
- The registers must be wide enough to store all the data corresponding to the lines that go through them.
- For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address.

Five stages of load instruction:

- The five stages of load instruction are the following:

1. Instruction fetch:

- The instruction is read from memory using the address in the PC and then being placed in the IF/ID pipeline register.
- The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.
- The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

2. Instruction decode and register file read:

- The IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.
- All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

3. Execute or address calculation:

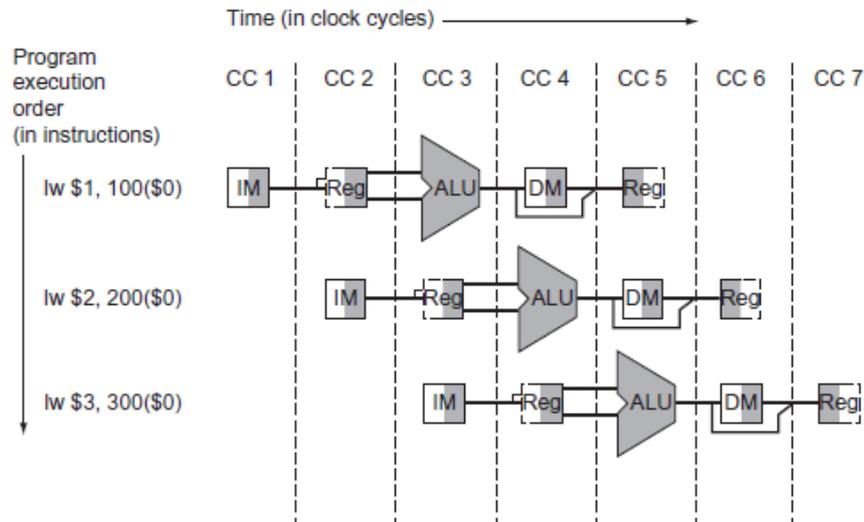
- The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU.
- That sum is placed in the EX/MEM pipeline register.

4. Memory access:

- The load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

5. Write-back:

- The final step is reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.



Five stages of store instruction:

1. Instruction fetch:

- The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified.

2. Instruction decode and register file read:

- The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register.

3. Execute and address calculation:

- In the third step, the effective address is placed in the EX/MEM pipeline register.

4. Memory access:

- The data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX.
- The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

5. Write-back:

- For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions.

Graphically Representing Pipelines

- Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single data path in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: multiple-clock-cycle pipeline diagrams.

- For example, consider the following five-instruction sequence:

lw \$10, 20(\$1)

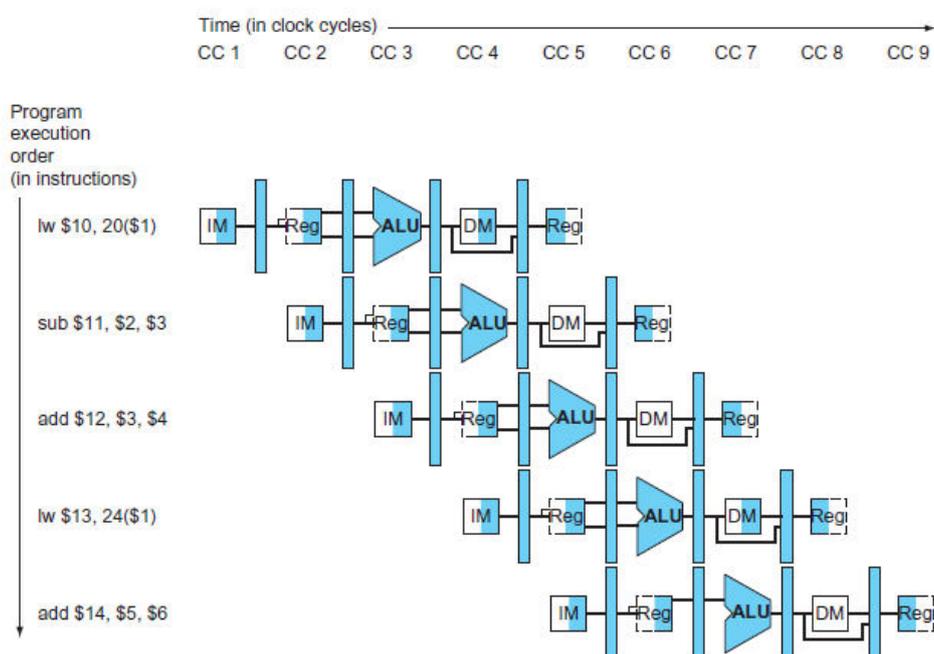
sub \$11, \$2, \$3

add \$12, \$3, \$4

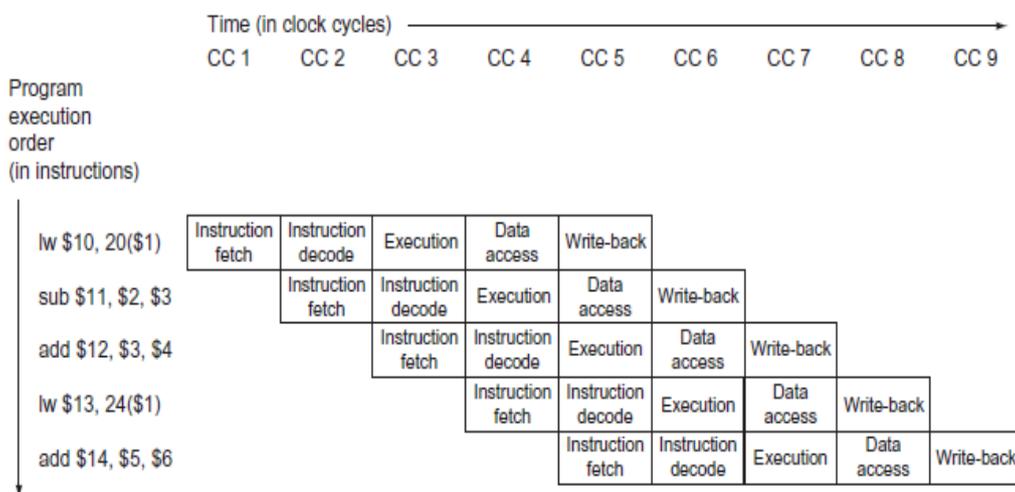
lw \$13, 24(\$1)

add \$14, \$5, \$6

- Figure shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page.



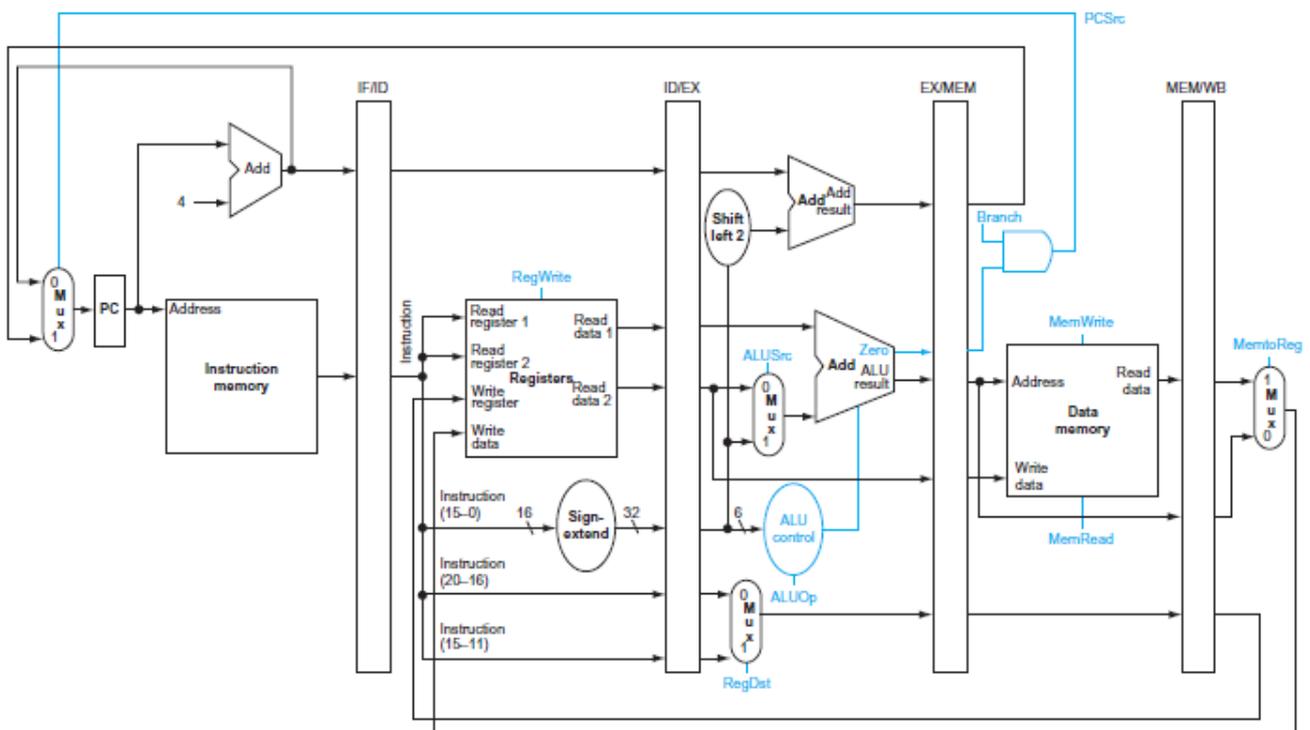
Multiple clock cycle pipeline diagrams



Traditional multiple-clock-cycle pipeline diagram of five instructions

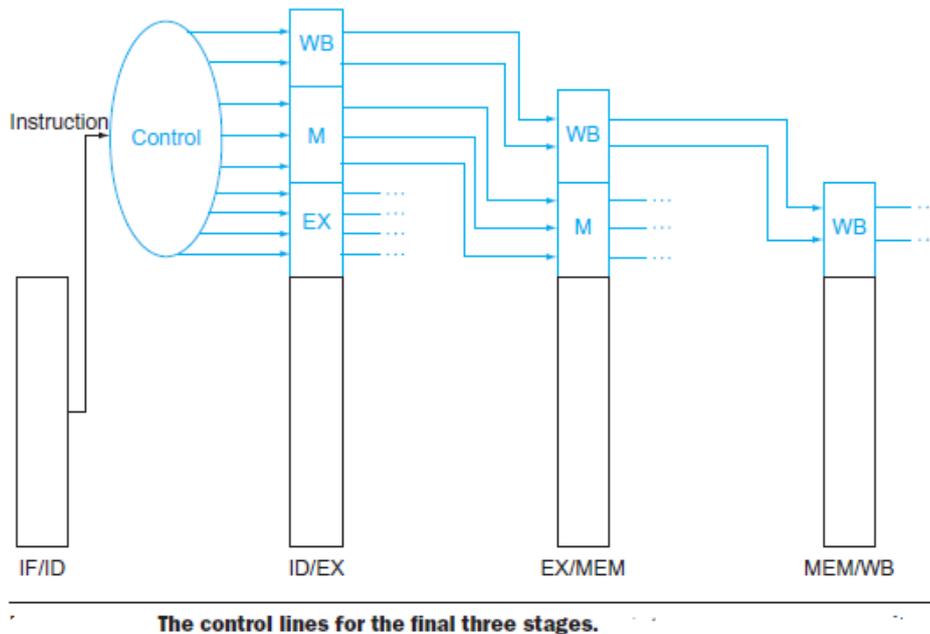
Pipelined Control

- Just as we add control to the pipelined data path. We can divide the control lines into five groups according to the pipeline stages.
1. **Instruction fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
 2. **Instruction decode/register file read:** As in the previous stage, the same thing happens at every clock cycle, so there are no optional control lines to set.
 3. **Execution/address calculation:** The signals to be set are RegDst, ALUOp, and ALUSrc. The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.
 4. **Memory access:** The control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal, load, and store instructions set these signals, respectively.
 5. **Write-back:** The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and Reg-Write, which writes the chosen value.



The pipelined data path with control signals

- Figure shows the control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the Sign-extend lines; three are used during the MEM stage, and the last two are passed to MEM/ WB for use in the WB stage.
- Since the control lines start with the EX stage, we can create the control information during instruction decode.



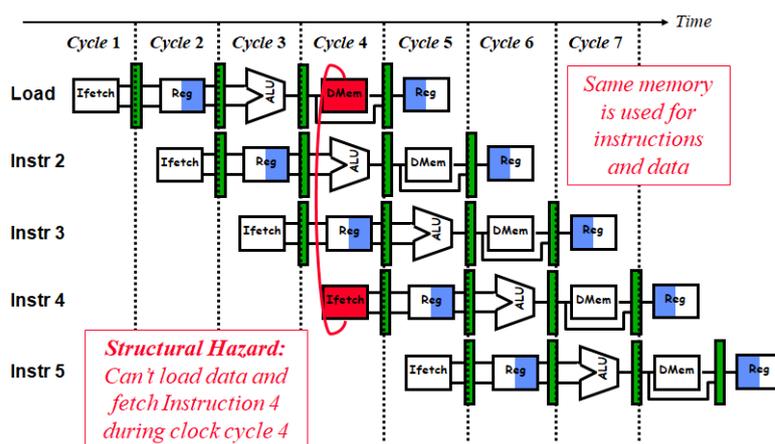
HANDLING DATA HAZARDS AND CONTROL HAZARDS:

Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.
 - Structural hazard
 - Data hazard
 - Control hazard

Structural hazard:

- The first hazard is called a structural hazard. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- The MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories.
- If the pipeline in Figure had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

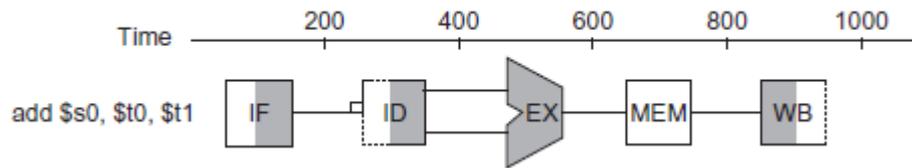


Data hazard:

- Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (\$s0):

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3

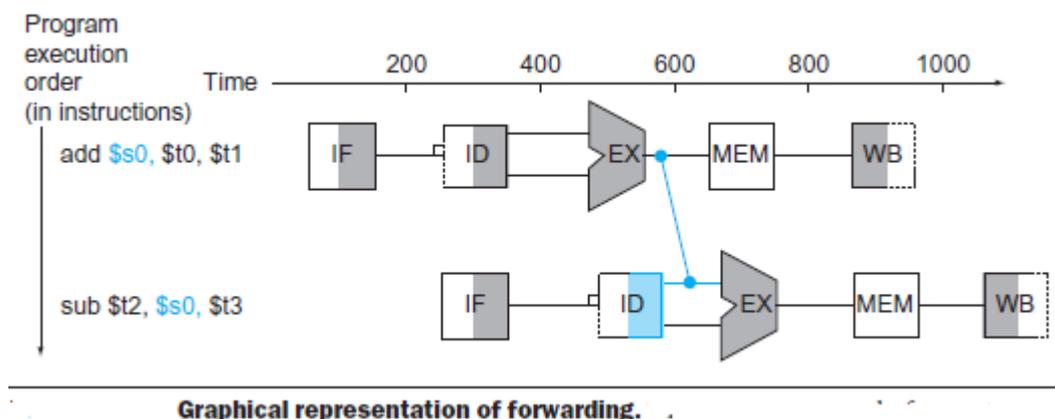


Graphical representation of the instruction pipeline

- In the above example, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.
- There are some solutions to rectify data hazard. They are
 - Forwarding or Bypassing
 - Hazard detection unit
 - Reordering code to avoid pipeline stalls
 - Hazard detection by software

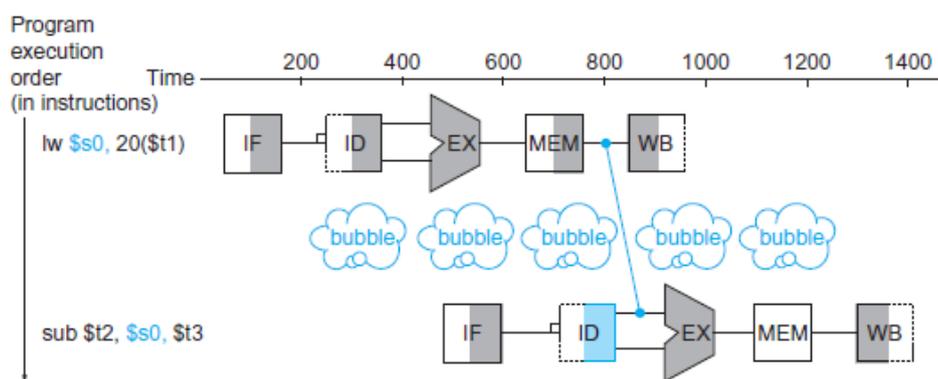
Forwarding or Bypassing:

- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.



Graphical representation of forwarding.

- The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register \$s0 read in the second stage of sub.
- Forwarding works very well but cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of \$s0 instead of an add. The desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub. Hence, even with forwarding, we would have to stall one stage for a **load-use data hazard**.
- It is a specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.



Stall even with forwarding

- Figure shows that we need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible.
- This figure shows an important pipeline concept, officially called a **pipeline stall**, but often given the nickname **bubble**.

Data Hazards: Forwarding versus Stalling

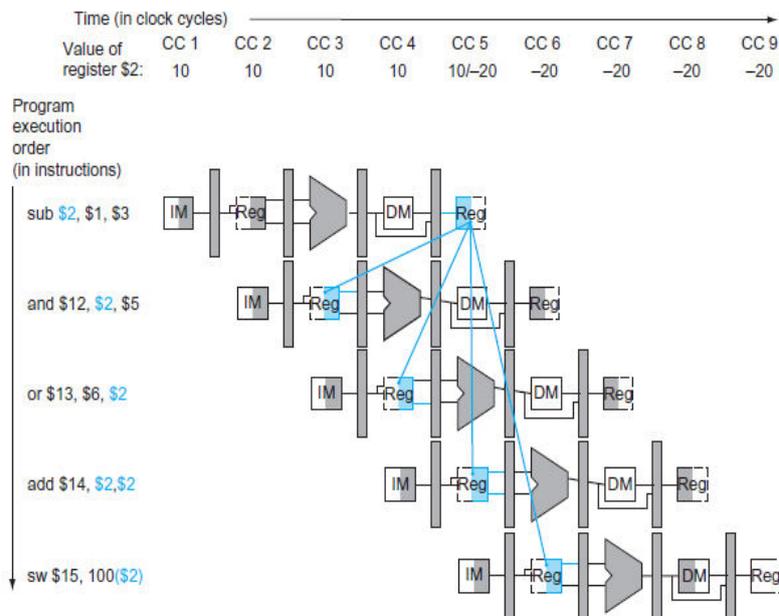
- Data hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- Consider the following instructions:

```

sub $2, $1,$3      # Register $2 written by sub
and $12,$2,$5     # 1st operand($2) depends on sub
or $13,$6,$2      # 2nd operand($2) depends on sub
add $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw $15,100($2)    # Base ($2) depends on sub

```

- The last four instructions are all dependent on the result in register \$2 of the first instruction. If register \$2 had the value 10 before the subtract instruction and -20 afterwards, the programmer intends that -20 will be used in the following instructions that refer to register \$2.
- Fig. Illustrates the execution of these instructions using a multiple clock cycle pipeline representation.



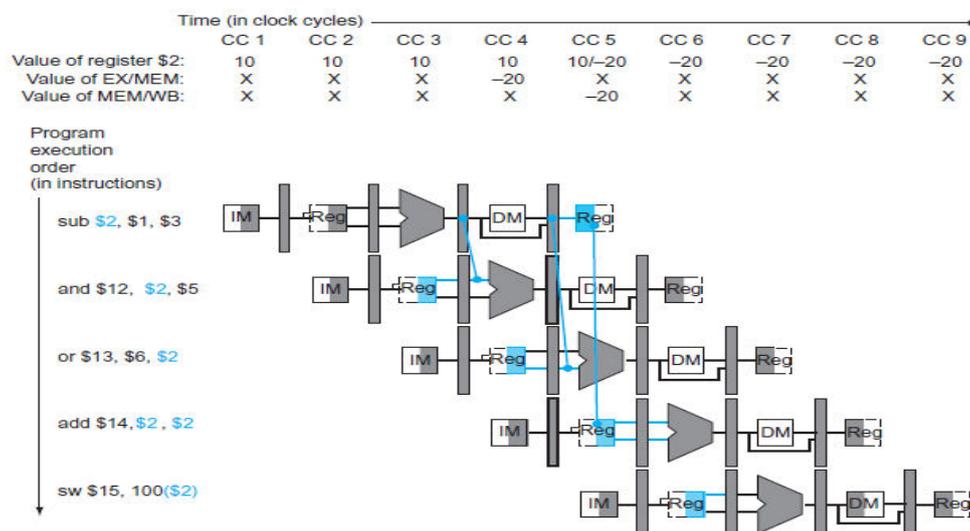
- In figure, the data from the sub instruction actually produced at the end of the EX stage or clock cycle 3. The data actually needed by the and & or instruction is the beginning of the EX stage, or clock cycles 4 & 5 respectively.
- Thus, we can execute this segment without stalls if we simply forward the data as soon as it is available to any units that need it before it is available to read from the register file.
- Data forwarding is effective to an operation in the EX stage, which may be either an ALU operation or an effective address calculation.
- This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage. We actually need the values as inputs to the ALU.
- Using this notation, the two pairs of hazard conditions are

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt



Data forwarding to avoid stalls

➤ Let's now write both the conditions for detecting hazards and the control signals to resolve them:

1. EX hazard:

if (EX/MEM.RegWrite

and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite

and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

2. MEM hazard:

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

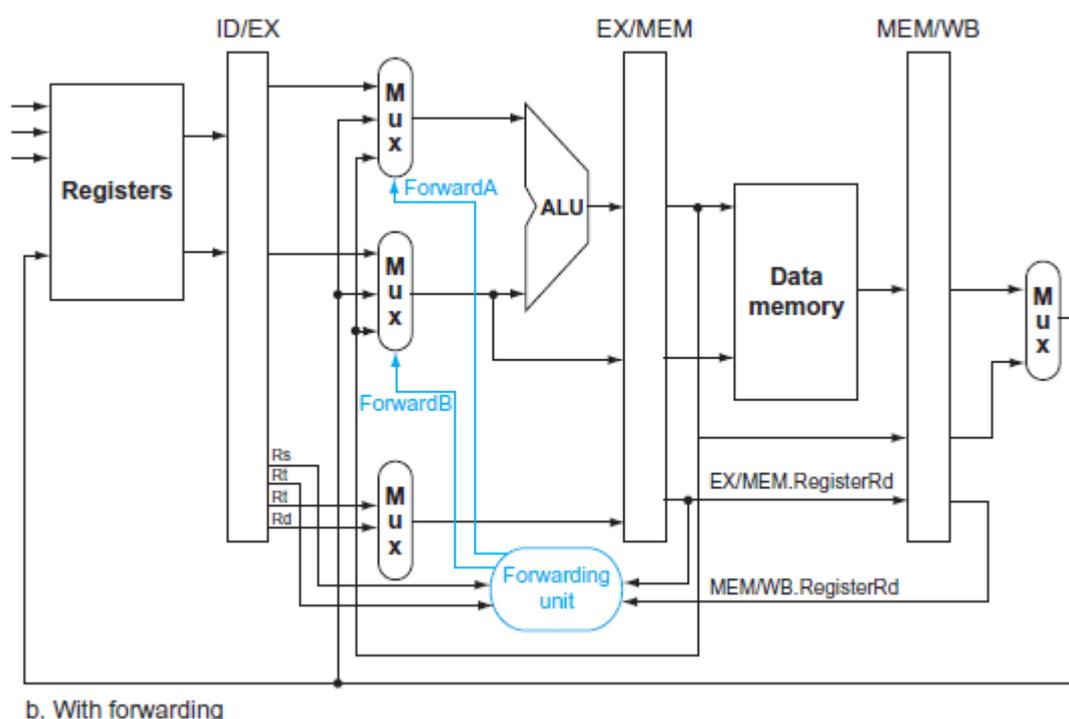
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

➤ As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.



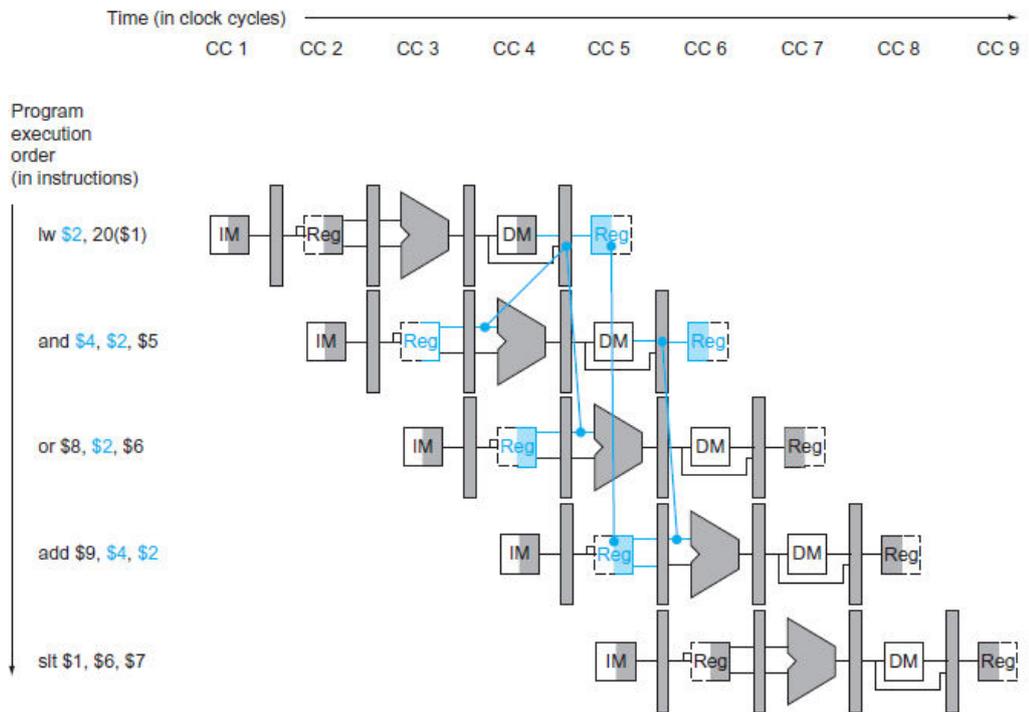
Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

The control values for the forwarding multiplexors

Data Hazards and Stalls:

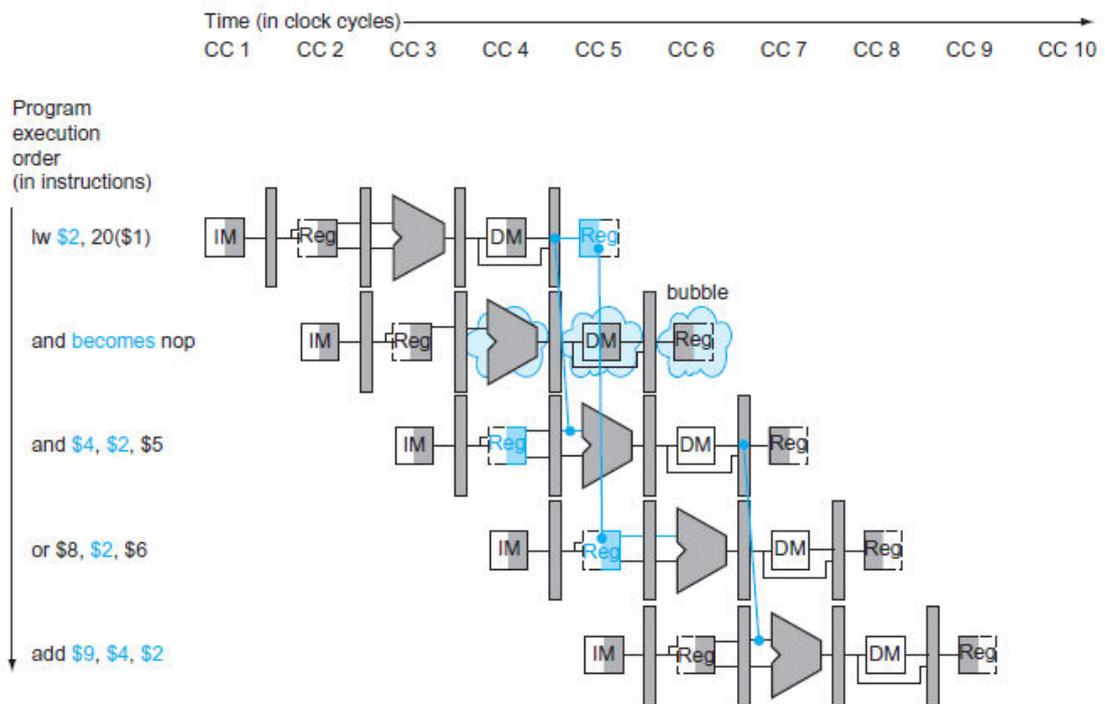
- The dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and its use. Checking for load instructions, the control for the hazard detection unit is this single condition:

if (ID/EX.MemRead and
 ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
 (ID/EX.RegisterRt = IF/ID.RegisterRt)))
 stall the pipeline

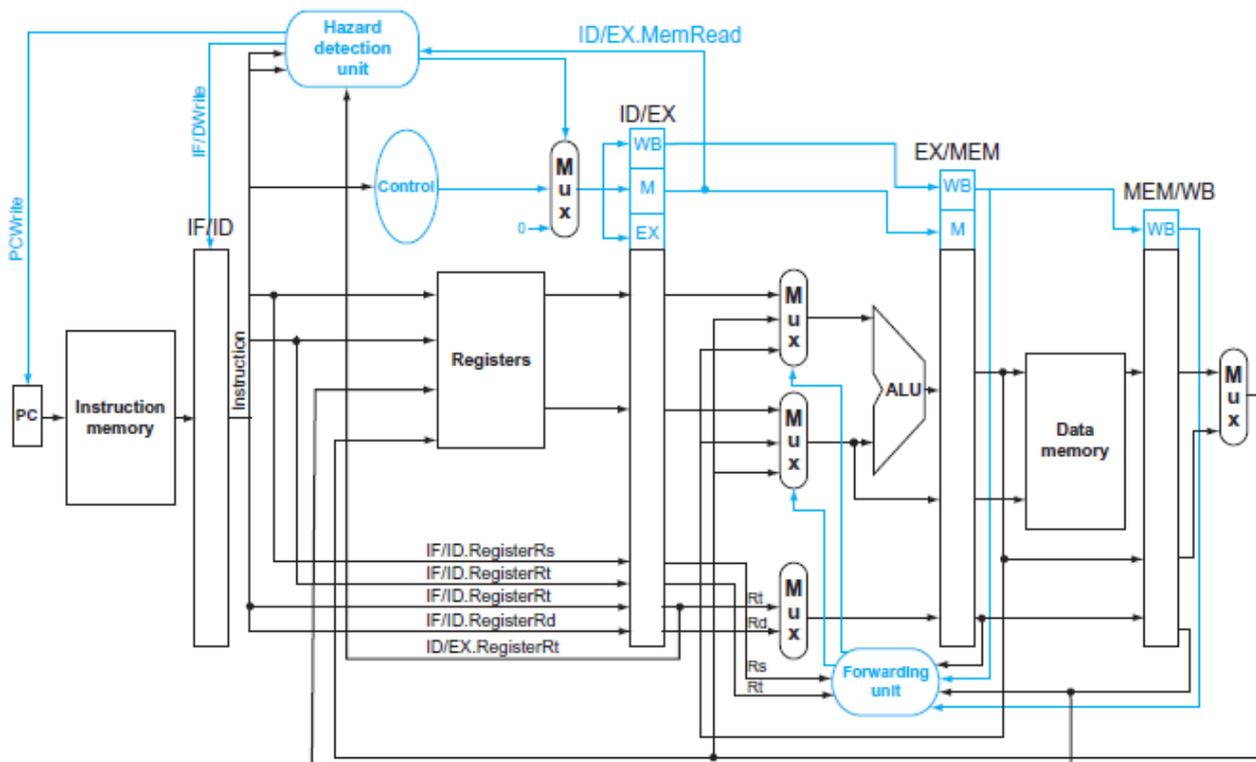


A pipelined sequence of instructions.

- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled, otherwise we would lose the fetched instruction. Below figure highlights the pipeline connections for both the hazard detection unit and the forwarding unit.
- The forwarding unit controls the ALU multipliers to replace the value from a general-purpose register with the value of the instruction.
- By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM and WB control fields of the ID/EX pipeline register to 0.



The way stalls are really inserted into the pipeline

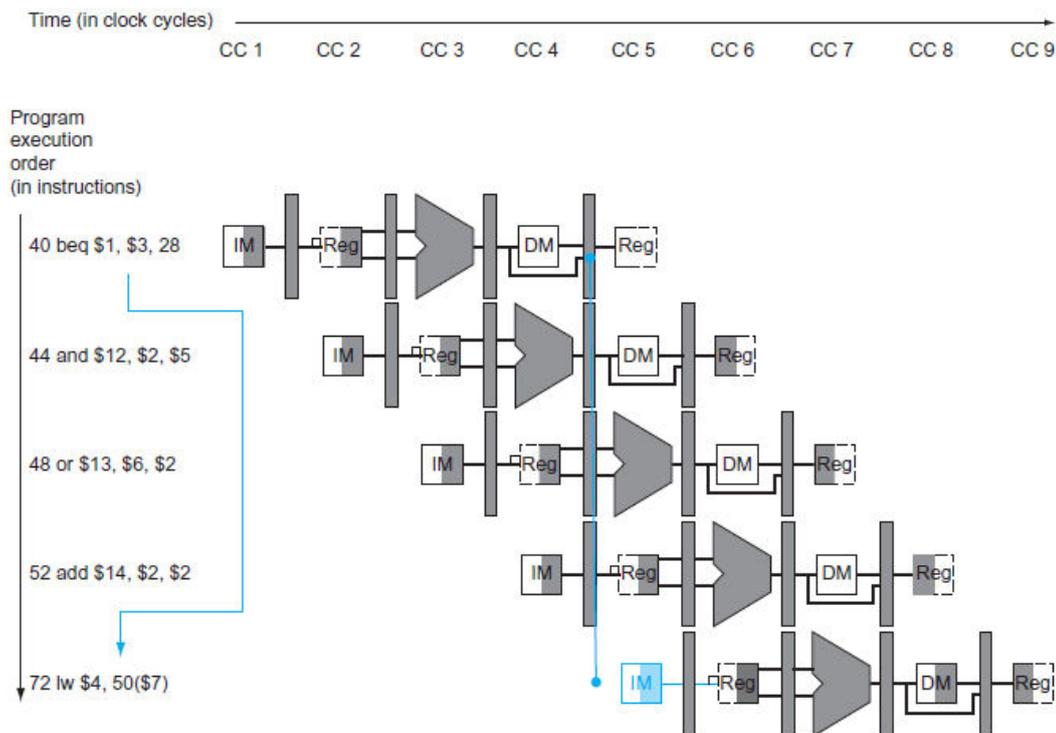


The hazard detection unit

- Figure highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s.

Control hazard:

- Control hazard is also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- Fig. shows a sequence of instructions and indicates when the branch would occur in this pipeline. An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage. This delay in determining the proper instruction to fetch is called a control hazard or branch hazard.

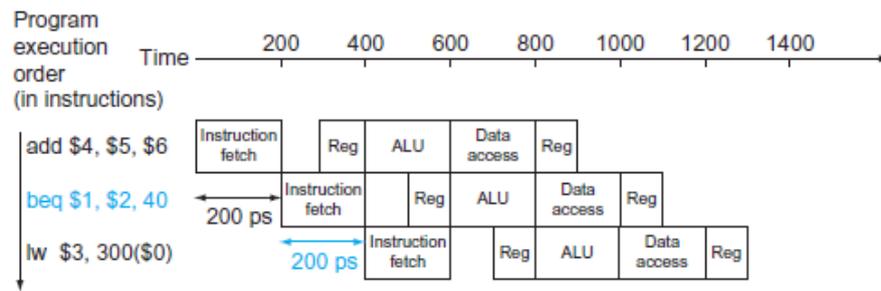


The impact of the pipeline on the branch instruction

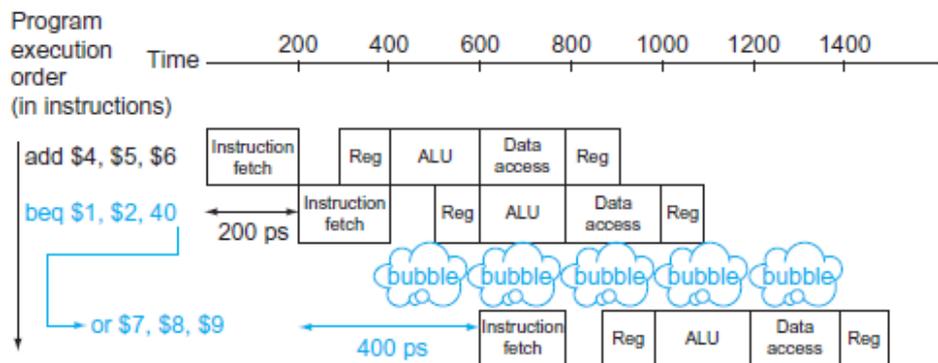
- This section on control hazards is shorter than the previous sections on data hazards. The reasons are that
 - control hazards are relatively simple to understand,
 - they occur less frequently than data hazards,
 - There is nothing as effective against control hazards as forwarding is against data hazards. Hence, we use simpler schemes.

Assume Branch Not Taken:

- A common improvement over branch stalling is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, the optimization halves the cost of control hazards.
- To discard instructions, we merely change the original control values to 0s. Discarding the instructions in the pipeline is called as flushing.



Pipelining that branches are not taken

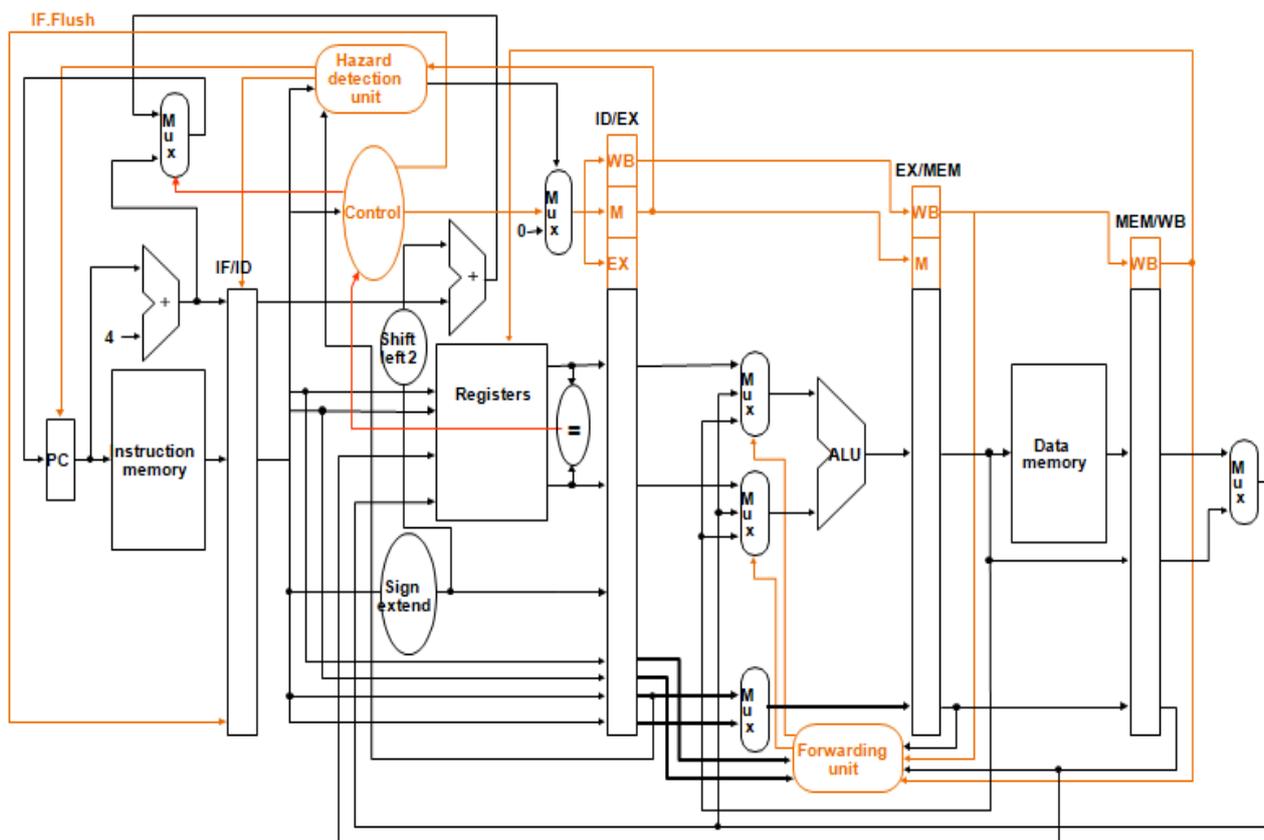


Pipelining that branch is taken

Reducing the Delay of Branches

- One way to improve branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed.
- The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty. When a more complex branch decision is required, a separate instruction that uses an ALU to perform a comparison is required.
- Moving the branch decision up requires two actions to occur earlier: **computing the branch target address and evaluating the branch decision.**
- The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch address from the EX stage to the ID stage; of course, the branch target address calculation will be performed for all instructions, but only used when needed.
- The harder part is the branch decision itself. For branch equal, we would compare the two registers read during the ID stage to see if they are equal. Equality can be tested by first exclusive ORing their respective bits and then ORing all the results. Moving the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization.

- Despite these difficulties, moving the branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched.
- To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a nop, an instruction that has no action and changes no state.



Dynamic Branch Prediction

- Assuming a branch is not taken is one simple form of branch prediction. In that case, if branch is taken, the pipeline is flushed. For the simple five-stage pipeline, it is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. It is possible to **predict** branch behaviour during program execution with more hardware.
- One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed and, if so, to begin fetching new instructions from the same place as the last time. This technique is called **dynamic branch prediction**.
- One implementation of that approach is a **branch prediction buffer** or **branch history table**. A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

- This is the simplest sort of buffer; if the prediction is the right one—it may have been put there by another branch that has the same low-order address bits. However, this doesn't affect correctness.
- Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

Implementation of branch prediction buffer:

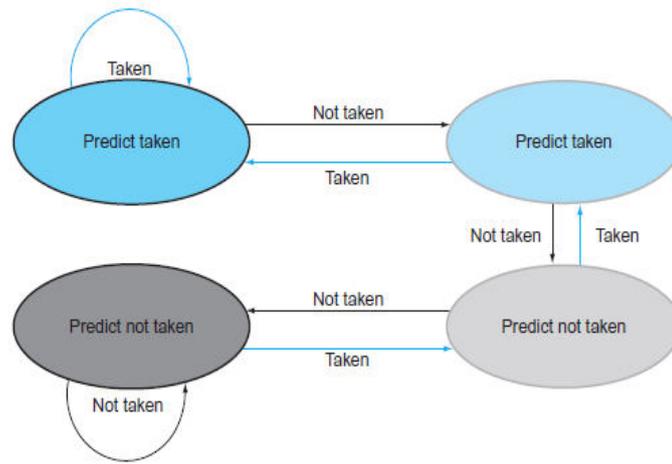
- A branch prediction buffer is also called branch history table. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not. There are three schemes to dynamic branch prediction.
 - Simple 1-bit prediction scheme
 - 2-bit prediction scheme
 - Tournament predictor

Simple 1-bit prediction scheme:

- If the bit is set prediction is branch taken, so fetching begins in the predicted direction. If the bit is not set, the prediction is branch not taken, so fetching continues in the same sequence.
- Prediction is just a hint that we hope is correct, so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.
- This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken.

2-bit prediction scheme:

- In a 2-bit scheme misprediction occurs only once. The 2 bits are used to encode the four states in the system.
- The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.



The states in a 2-bit prediction scheme

Tournament predictor:

- A more recent innovation in branch prediction is the use of tournament predictors. A tournament predictor uses multiple predictors for each branch. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behaviour.
- A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor, favouring whichever of the two predictors has been more accurate. Some recent microprocessors use such elaborate predictors.

EXCEPTIONS:

- Exceptions are also called **interrupt**. It is an unscheduled event that disrupts program execution; used to detect overflow. Events other than branches or jumps that change the normal flow of instruction execution come under exception.
- They were initially created to handle unexpected events from within the processor, like arithmetic overflow.
- In MIPS convention, the term exception to refer to any unexpected change in control flow without distinguishing whether the cause is internal or external; we use the term interrupt only when the event is externally caused.
- Here are five examples showing whether the situation is internally generated by the processor or externally generated:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

- Detecting exceptional conditions and taking the appropriate action is often on the critical timing path of a processor, which determines the clock cycle time and thus performance.

Exceptions handling in MIPS architecture:

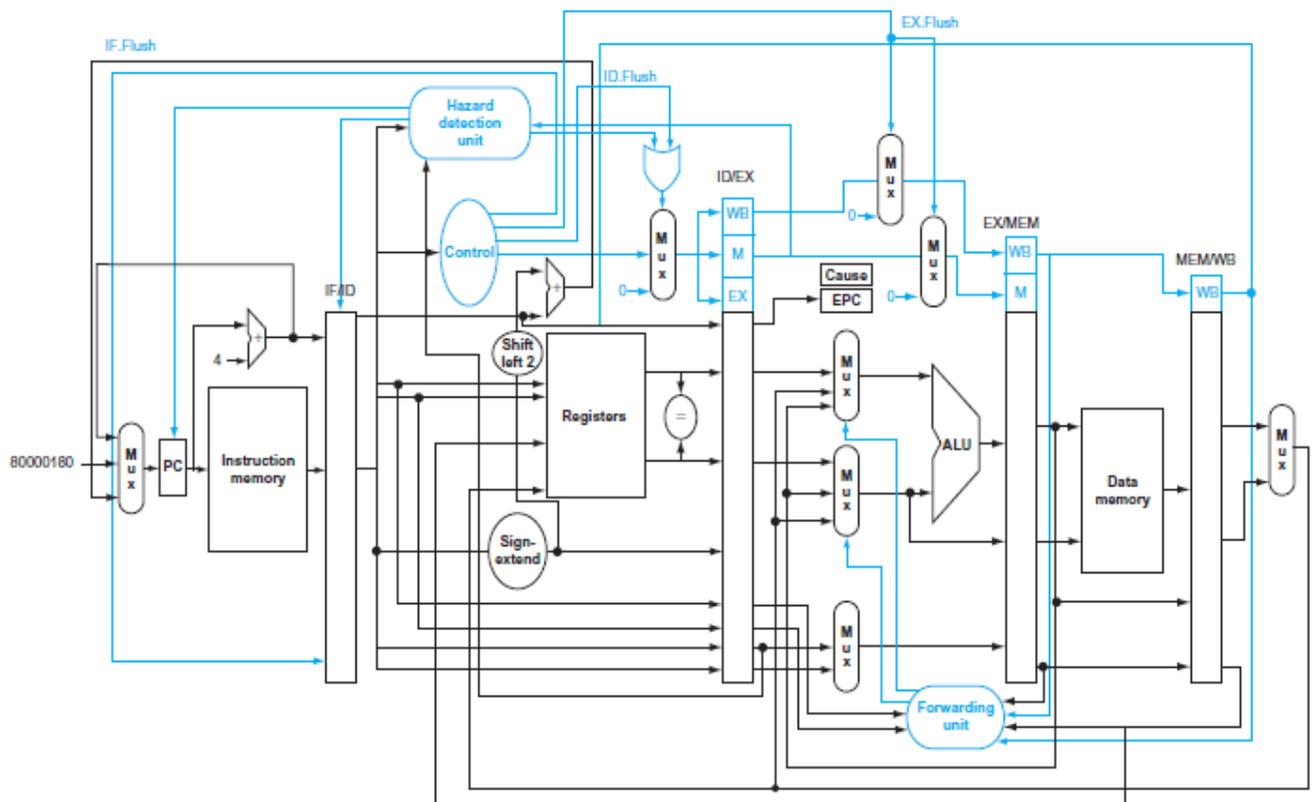
- The two types of exceptions that our current implementation can generate are execution of an **undefined instruction** and an **arithmetic overflow**. The basic action that the processor must perform when an exception occurs is to save the address of the offending instruction in the exception program counter (EPC) and then transfer control to the operating system at some specified address.
- The operating system can then take the appropriate action, which may involve providing some service to the user program, taking some predefined action in response to an overflow, or stopping the execution of the program and reporting an error.
- After performing whatever action is required because of the exception, the operating system can terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.
- For the operating system to handle the exception, it must know the reason for the exception, in addition to the instruction that caused it. There are two main methods used to communicate the reason for an exception.
- **Status register (called the Cause register)** - which holds a field that indicates the reason for the exception.
- **Vectored interrupts**- In a vectored interrupt; the address to which control is transferred is determined by the cause of the exception.
- For example, to accommodate the two exception types listed above, we might define the following two exception vector addresses:

Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000 _{hex}
Arithmetic overflow	8000 0180 _{hex}

- The operating system knows the reason for the exception by the address at which it is initiated. When the exception is not vectored, a single entry point for all exceptions can be used, and the operating system decodes the status register to find the cause.
- We can implement exception handling by adding the following two additional registers to our simple MIPS implementation:
 - **EPC:** A 32-bit register used to hold the address of the affected instruction.
 - **Cause:** A register used to record the cause of the exception. It is also this 32 bits wide.

Exceptions in a Pipelined Implementation

- A pipelined implementation treats exceptions as another form of control hazard. For example, suppose there is an arithmetic overflow in an add instruction. It is handled by flushing the instructions that follow the add instruction from the pipeline and begin fetching instructions from the new address.
- In our example, exception vector address for arithmetic overflow is $8000\ 0180_{\text{hex}}$. So, the PC must be loaded with this new address $8000\ 0180_{\text{hex}}$. That can be done by additional input to the PC multiplexor. There are two types of exceptions,
 - **Imprecise exception**- Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
 - **Precise exception** - An interrupt or exception that is always associated with the correct instruction in pipelined computers.



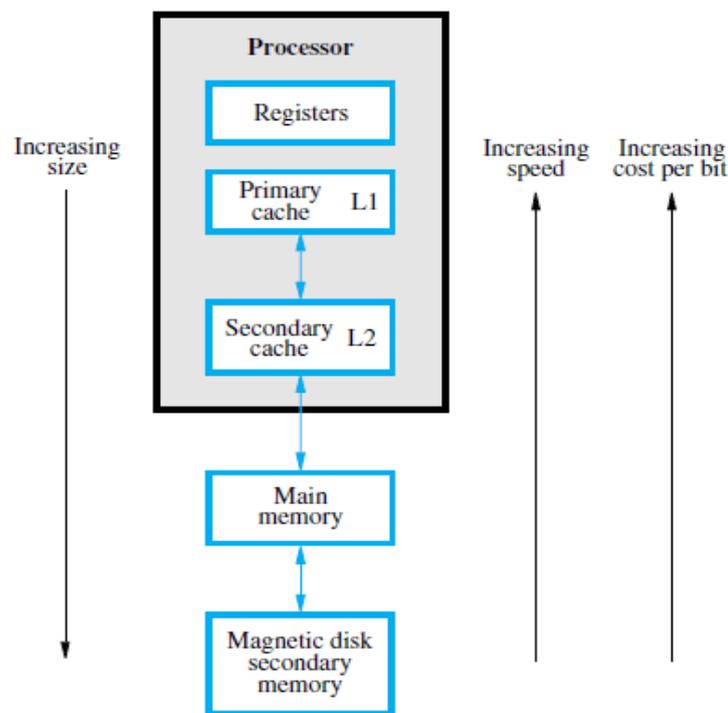
The data path with controls to handle exceptions

UNIT V
MEMORY AND I/O

Memory Concepts and Hierarchy – Memory Management – Cache Memories: Mapping and Replacement Techniques – Virtual Memory – DMA – I/O – Accessing I/O: Parallel and Serial Interface – Interrupt I/O – Interconnection Standards: USB, SATA

MEMORY HIERARCHY:

- An ideal memory would be fast, large, and inexpensive. It is clear that a very fast memory can be implemented using static RAM chips. But, these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than dynamic RAM cells.
- Although dynamic memory units with gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data.
- A solution is provided by using secondary storage, mainly magnetic disks, to provide the required memory space. Disks are available at a reasonable cost, and they are used extensively in computer systems. However, they are much slower than semiconductor memory units.
- In summary, a very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main memory can be built with dynamic RAM technology. This leaves the more expensive and much faster static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories.
- All of these different types of memory units are employed effectively in a computer system. The entire computer memory can be viewed as the hierarchy depicted in Figure The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of speed of access. Of course, the registers provide only a minuscule portion of the required memory.



Memory hierarchy

- At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a processor cache, holds copies of the instructions and data stored in a much larger memory that is provided externally.
- There are often two or more levels of cache. A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers. The primary cache is referred to as the level 1 (L1) cache. A larger, and hence somewhat slower, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the level 2 (L2) cache. Often, the L2 cache is also housed on the processor chip.
- Some computers have a level 3 (L3) cache of even larger size, in addition to the L1 and L2 caches. An L3 cache, also implemented in SRAM technology, may or may not be on the same chip with the processor and the L1 and L2 caches.
- The next level in the hierarchy is the main memory. This is a large memory implemented using dynamic memory components, typically assembled in memory modules such as DIMMs.
- The main memory is much larger but significantly slower than cache memories. In a computer with a processor clock of 2 GHz or higher, the access time for the main memory can be as much as 100 times longer than the access time for the L1 cache.
- Disk devices provide a very large amount of inexpensive memory, and they are widely used as secondary storage in computer systems. They are very slow compared to the main memory. They represent the bottom level in the memory hierarchy.
- If the data requested by the processor appears in some block in the upper level, this is called a **hit**. If the data is not found in the upper level, the request is called a **miss**.
- The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. The **hit rate, or hit ratio, is the fraction of memory accesses** found in the upper level; it is often used as a measure of the performance of the memory hierarchy.
- The **miss rate (1–hit rate) is the fraction of memory accesses** not found in the upper level.
- **Hit time is the time to access the upper level** of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk).
- **Miss penalty: The time** required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

MEMORY TECHNOLOGIES:

- There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM (static random access memory).
- DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon.
- The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices.
- The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012:

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

SRAM Technology

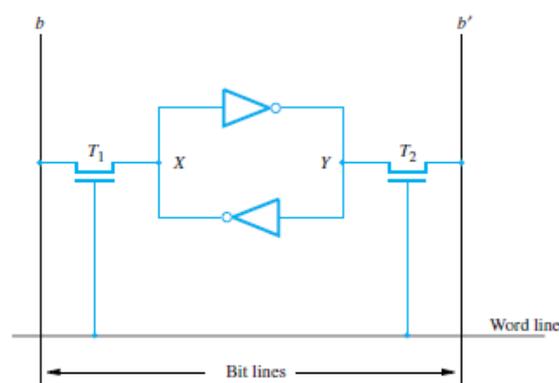
- SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write.
- SRAMs have a fixed access time to any datum, though the read and write access times may differ.
- SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read.
- SRAM needs only minimal power to retain the charge in standby mode.
- Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories.
- Figure illustrates how a static RAM (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors T₁ and T₂. These transistors act as switches that can be opened or closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state.

Read Operation

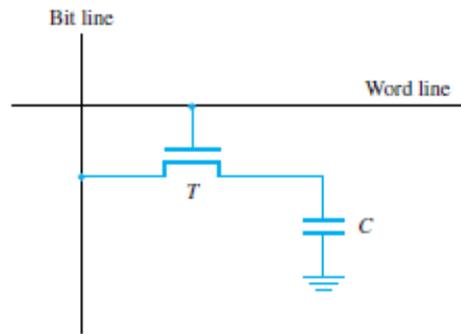
- In order to read the state of the SRAM cell, the word line is activated to close switches T₁ and T₂. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. Thus, b and b' are always complements of each other. The Sense/Write circuit at the end of the two bit lines monitors their state and sets the corresponding output accordingly.

Write Operation

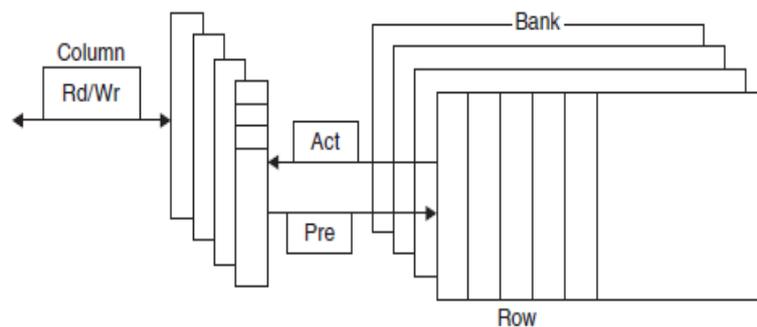
- During a Write operation, the Sense/Write circuit drives bit lines b and b', instead of sensing their state. It places the appropriate value on bit line b and its complement on b' and activates the word line. This forces the cell into the corresponding state, which the cell retains when the word line is deactivated.



A static RAM cell

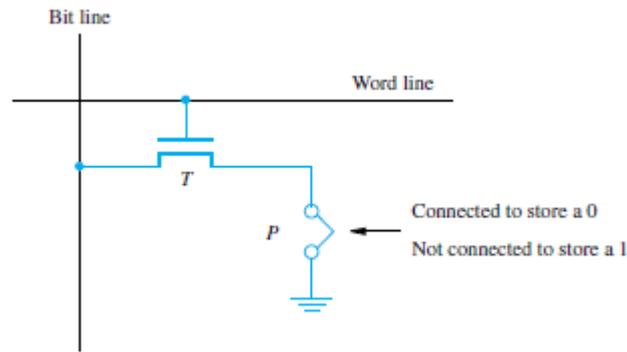


- Figure shows the internal organization of a DRAM, Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows.
- Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer.
- When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address.
- Each command, as well as block transfers, is synchronized with a clock.



ROM

- A memory is called a read-only memory, or ROM, when information can be written into it only once at the time of manufacture. Figure shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point *P*; otherwise, a 1 is stored.
- The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated to close the transistor switch. As a result, the voltage on the bit line drops to near zero if there is a connection between the transistor and ground.
- If there is no connection to ground, the bit line remains at the high voltage level, indicating a 1. A sense circuit at the end of the bit line generates the proper output value.
- The state of the connection to ground in each cell is determined when the chip is manufactured, using a mask with a pattern that represents the information to be stored.



A ROM cell

PROM

- Some ROM designs allow the data to be loaded by the user, thus providing a programmable ROM (PROM). Programmability is achieved by inserting a fuse at point P in Figure. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible.
- PROMs provide flexibility and convenience not available with ROMs. The cost of preparing the masks needed for storing a particular information pattern makes ROMs cost effective only in large volumes. The alternative technology of PROMs provides a more convenient and considerably less expensive approach, because memory chips can be programmed directly by the user.

EPROM

- Another type of ROM chip provides an even higher level of convenience. It allows the stored data to be erased and new data to be written into it. Such an erasable, reprogrammable ROM is usually called an EPROM. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs or PROMs while software is being developed.
- In this way, memory changes and updates can be easily made. An EPROM cell has a structure similar to the ROM cell in Figure. However, the connection to ground at point P is made through a special transistor. The transistor is normally turned off, creating an open switch. It can be turned on by injecting charge into it that becomes trapped inside.
- Erasure requires dissipating the charge trapped in the transistors that form the memory cells. This can be done by exposing the chip to ultraviolet light, which erases the entire contents of the chip. To make this possible, EPROM chips are mounted in packages that have transparent windows.

EEPROM

- An EPROM must be physically removed from the circuit for reprogramming. Also, the stored information cannot be erased selectively. The entire contents of the chip are erased when exposed to ultraviolet light.
- Another type of erasable PROM can be programmed, erased, and reprogrammed electrically. Such a chip is called an electrically erasable PROM, or EEPROM. It does not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively.
- One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity. However, this disadvantage is outweighed by the many advantages of EEPROMs. They have replaced EPROMs in practice.

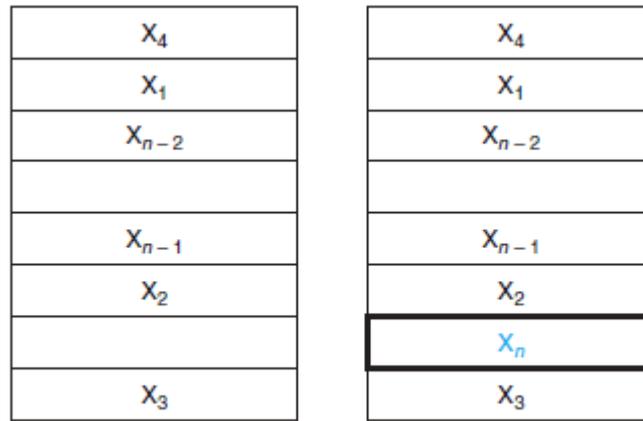
Flash Memory

- An approach similar to EEPROM technology has given rise to *flash memory* devices. A flash cell is based on a single transistor controlled by trapped charge, much like an EEPROM cell. Also like an EEPROM, it is possible to read the contents of a single cell.
- The key difference is that, in a flash device, it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased.
- Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.
- The low power consumption of flash memories makes them attractive for use in portable, battery-powered equipment. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players.
- In hand-held computers and cell phones, a flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. A flash memory is used in digital cameras to store picture data.
- In MP3 players, flash memories store the data that represent sound. Cell phones, digital cameras, and MP3 players are good examples of embedded systems.
- Larger memory modules consisting of a number of chips are used where needed. There are two popular choices for the implementation of such modules: flash cards and flash drives

CACHE MEMORY:

- The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called locality of reference.
- If programs are executed repeatedly during some time period, this behaviour manifests itself in two ways: temporal and spatial.
- Figure shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X_1, X_2, \dots, X_{n-1} , and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.
- If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the address of the word in memory.
- This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct mapped cache is usually simple.
- For example, almost all direct-mapped caches use this mapping to find a block:

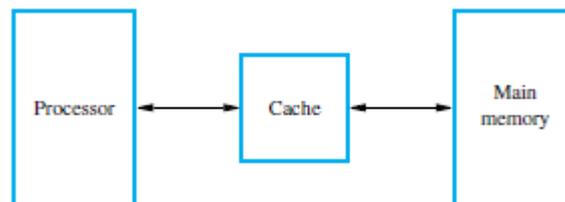
$$(\text{Block address}) \text{ modulo } (\text{Number of blocks in the cache})$$



a. Before the reference to X_n b. After the reference to X_n

The cache just before and just after a reference to a word X_n that is not initially in the cache

- Consider the arrangement in Figure. When the processor issues a Read request, the contents of a block of memory words containing the location specified are transferred into the cache. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache.
- Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.
- When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache’s replacement algorithm.



Use of a cache memory

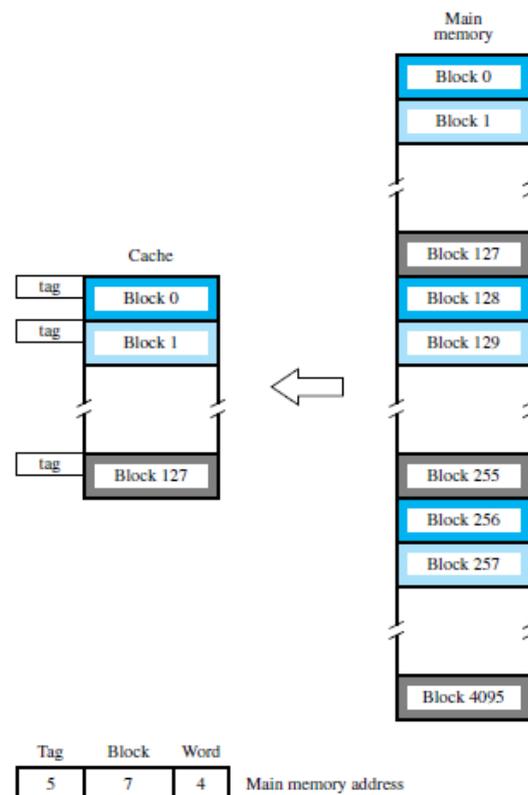
Mapping Functions

- There are several possible methods for determining where memory blocks are placed in the cache. It is instructive to describe these methods using a specific small example.
- Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address.
- The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we have assumed that consecutive addresses refer to consecutive words.

Direct Mapping

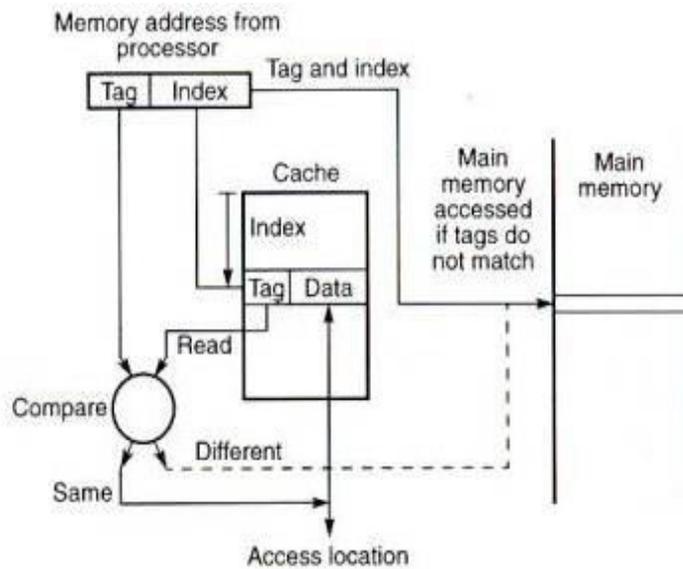
- The simplest way to determine cache locations in which to store memory blocks is the direct-mapping technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure.
- Thus, whenever one of the main memory blocks 0, 128, 256 . . . are loaded into the cache, it is stored in cache block 0.

- Blocks 1, 129, 257 . . . are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full.
- For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.
- With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure. The low-order 4 bits select one of 16 words in a block.
- When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the block are stored in 5 tag bits associated with its location in the cache.
- The tag bits identify which of the 32 main memory blocks mapped into this cache position is currently resident in the cache.



Direct-mapped cache

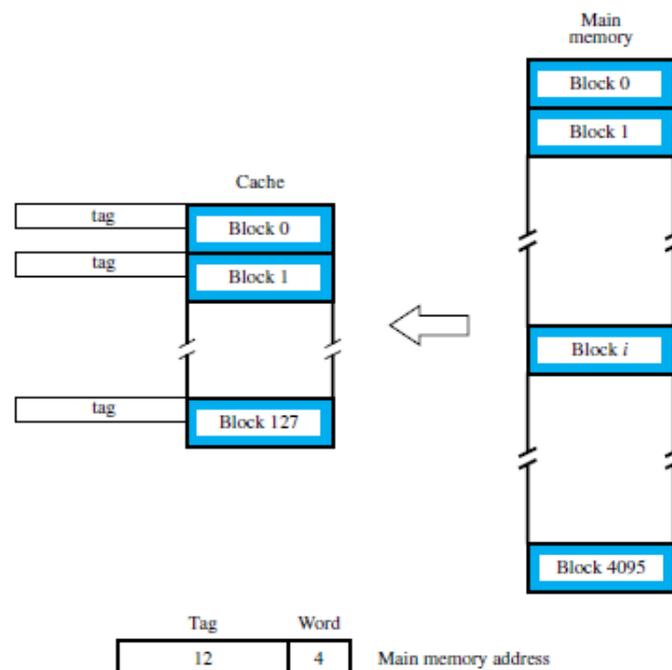
- As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache.
- The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache.
- If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.
- Consider the example shown in figure. The address from the processor is divided in to two fields, a tag and an index. The tag consists of the higher significant bits of the address, which are stored with the data. The index is the lower significant bits of the address used to address the cache.
- When the memory is referenced, the index is first used to access a word in the cache. Then the tag stored in the accessed word is read and compared with the tag in the address. If the two tags are the same, indicating that the word is the one required, access is made to the addressed cache word.
- If the tags are not the same, indicating that the required word is not in the cache, reference is made to the main memory to find it.



Cache with direct mapping

Associative Mapping

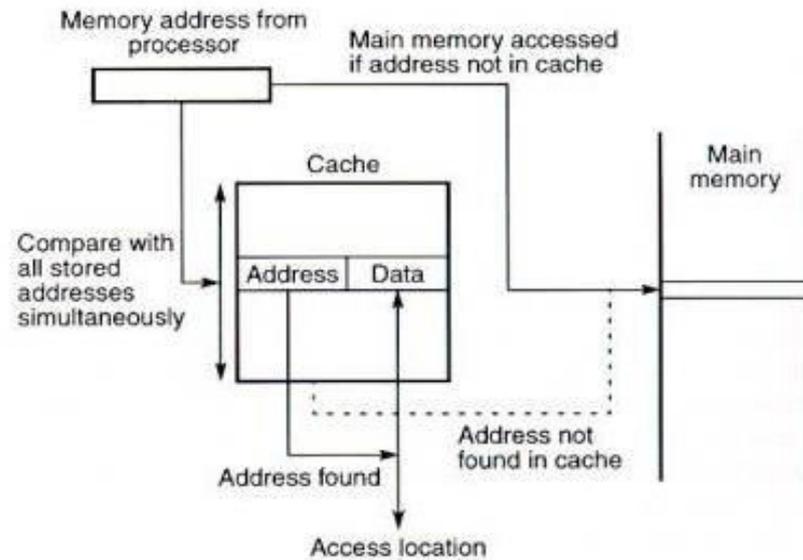
- Figure shows the most flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache.
- The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the associative-mapping technique. It gives complete freedom in choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache.



Associative-mapped cache

- When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible.

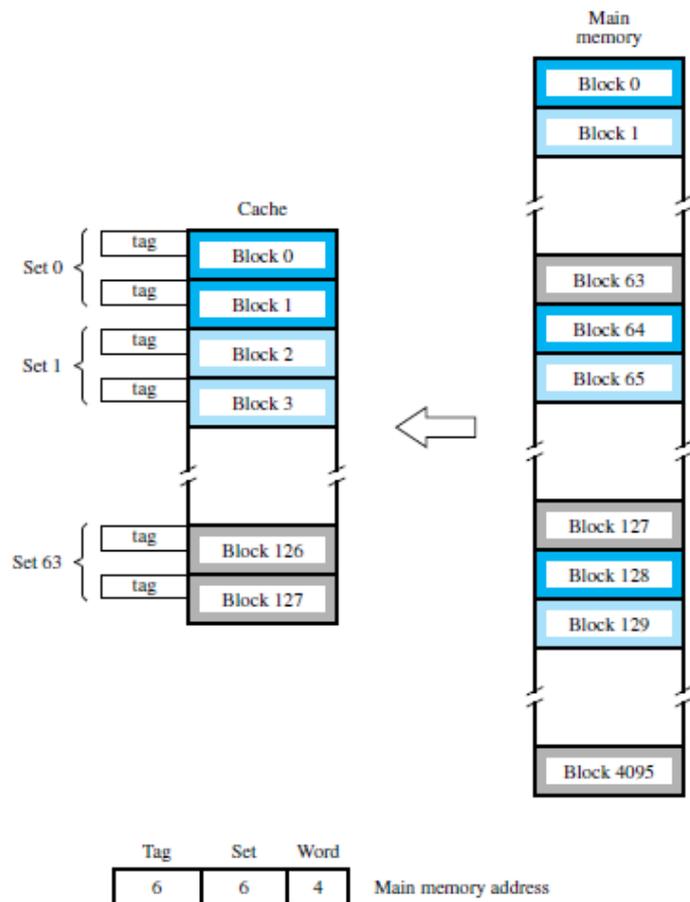
- The complexity of an associative cache is higher than that of a direct-mapped cache, because of the need to search all 128 tag patterns to determine whether a given block is in the cache. To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an associative search.
- A fully associative cache requires the cache to be composed of associative memory holding both the memory address and the data for each cached line. The incoming memory address is simultaneously compared with all stored addresses using the internal logic of associative memory, as shown in figure.
- If the match is found, the corresponding data is read out. Single words from anywhere within the main memory could be held in cache, if the associative part of the cache is capable of holding a full address.



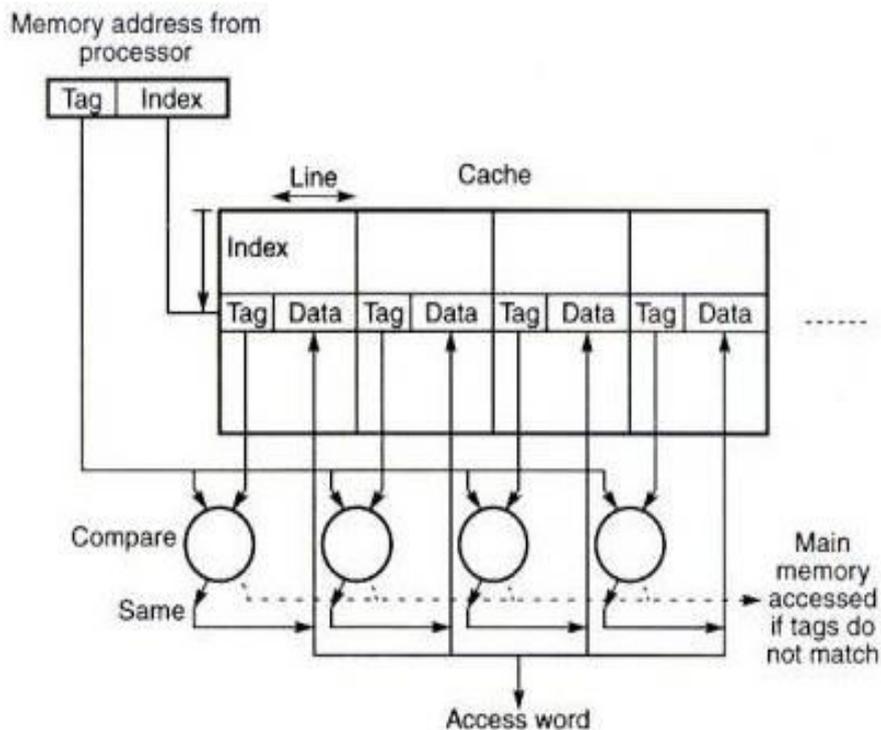
Cache with fully associative mapping

Set-Associative Mapping

- Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search.
- An example of this set-associative-mapping technique is shown in Figure for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block.
- The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.
- The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on.
- The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k -way set-associative cache.



Set-associative-mapped cache with two blocks per set



Cache with set associative mapping

- The cache is divided into sets of blocks. Each block in each set has a stored tag which, together with the index, completes the identification of the block. First, the index of the address from the processor is used to access the set. Then, comparators are used to compare all the tags of the selected set with the incoming tag. If a match is found, the corresponding location is accessed, otherwise, as before; an access to the main memory is made.

Replacement Algorithms

- In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial.
- In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite.
- In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced.
- Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used (LRU) block, and the technique is called the LRU replacement algorithm

MEASURING AND IMPROVING CACHE PERFORMANCE:

- In this section, we begin by examining ways to measure and analyze cache performance. We then explore two different techniques for improving cache performance.
- One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location.
- The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called multilevel caching.
- CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

- The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.
- Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

- The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

- Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$$

- If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

- We can also factor this as

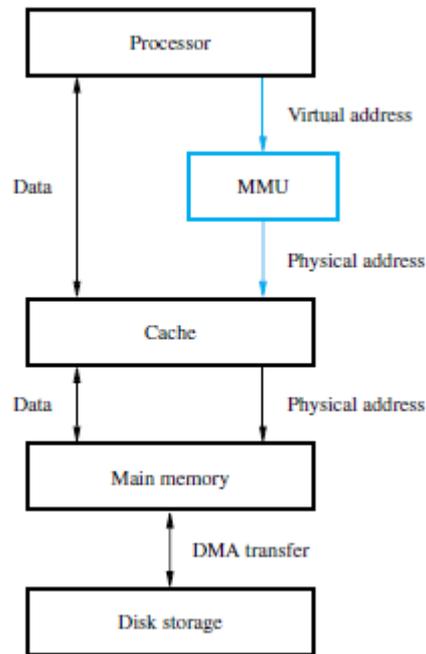
$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

- To capture the fact that the time to access data for both hits and misses affects performance; designers sometime use average memory access time (AMAT) as a way to examine alternative cache designs.
- Average memory access time is the average time to access memory considering both hits and misses and the frequency of different accesses; it is equal to the following:

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

VIRTUAL MEMORY:

- In most modern computer systems, the physical main memory is not as large as the address space of the processor. For example, a processor that issues 32-bit addresses has an addressable space of 4G bytes. The size of the main memory in a typical computer with a 32-bit processor may range from 1G to 4G bytes.
- If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as virtual memory.
- Application programmers need not be aware of the limitations imposed by the available main memory. They prepare programs using the entire address space of the processor.
- Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called virtual or logical addresses. These addresses are translated into physical addresses by a combination of hardware and software actions.
- If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.
- Figure shows a typical organization that implements virtual memory. A special hardware unit, called the Memory Management Unit (MMU), keeps track of which parts of the virtual address space are in the physical memory.
- When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory.



Virtual memory organization

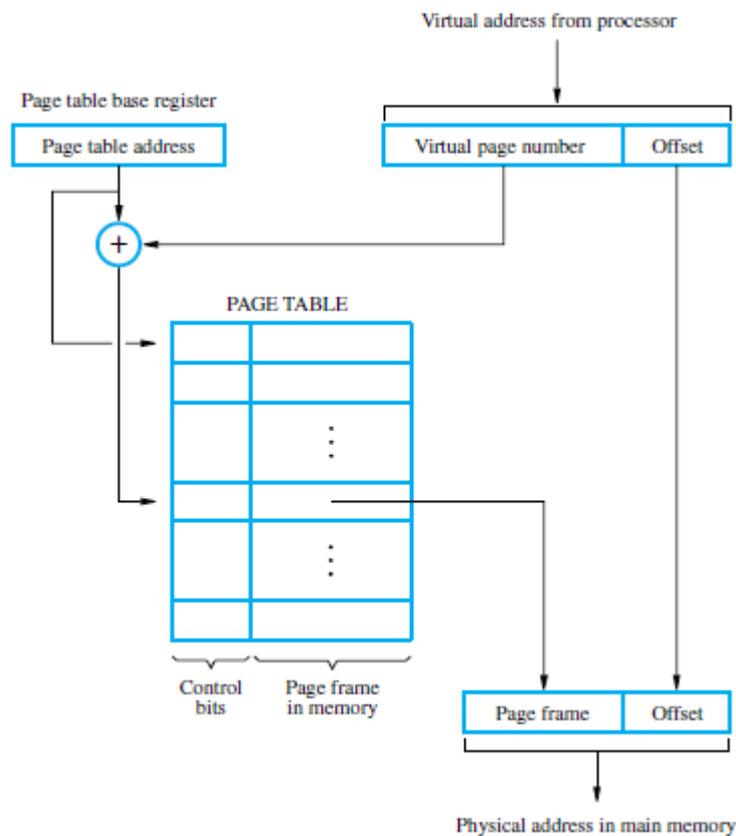
Address Translation

- A virtual-memory address-translation method based on the concept of fixed-length pages is shown schematically in Figure. Each virtual address generated by the processor is interpreted as a virtual page number (high-order bits) followed by an offset (low-order bits) that specifies the location of a particular byte (or word) within a page.
- Information about the main memory location of each page is kept in a **page table**. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a **page frame**.
- The starting address of the page table is kept in a page table base register. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.
- Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory.
- One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. It allows the operating system to invalidate the page without actually removing it. (Valid bit)
- Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. (Modify bit)
- Other control bits indicate various restrictions that may be imposed on accessing the page. For example, a program may be given full read and write permission, or it may be restricted to read accesses only. (Access control bits)

The following steps are used in address translation:

1. Virtual address generated by the processor is divided into two parts as virtual page number and offset.
2. The virtual page number in the address is added with the contents of page table base register to get the required entry in the page table.

3. The page table entry provides the address of page in memory.
4. Physical address is generated by combining the page frame address with the offset field of the virtual address.



Virtual-memory address translation

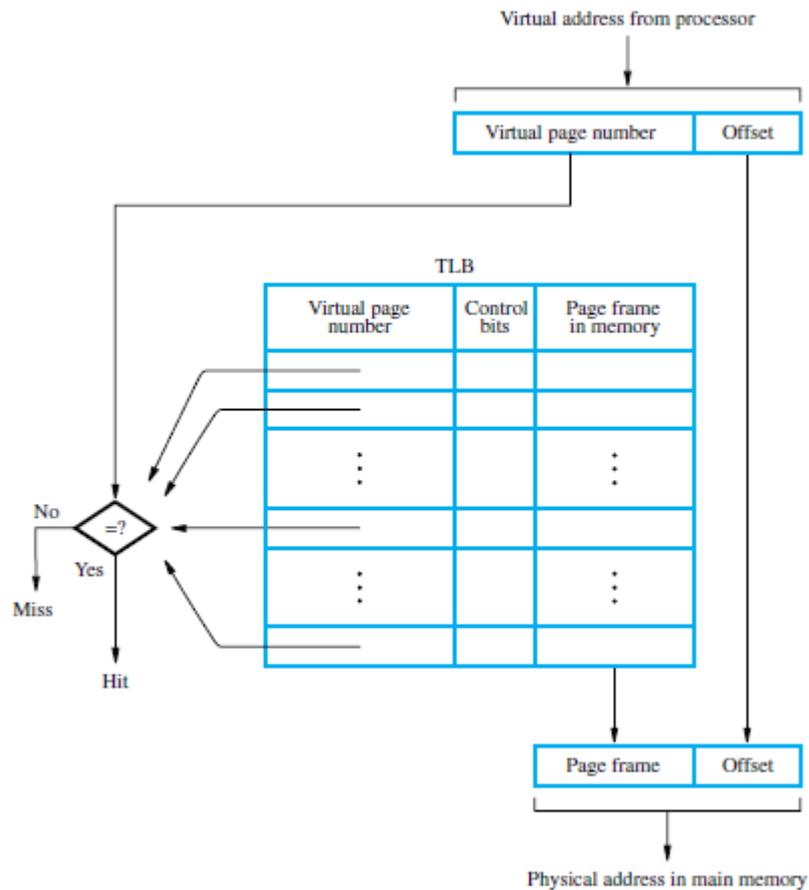
TRANSLATION LOOKASIDE BUFFER

- The page table information is used by the MMU for every read and write access. Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory.
- The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the Translation Lookaside Buffer (TLB).
- The TLB functions as a cache for the page table in the main memory. Each entry in the TLB includes a copy of the information in the corresponding entry in the page table. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page. Figure shows a possible organization of a TLB that uses the associative-mapping technique. Set-associative mapped TLBs are also found in commercial products.

Address translation proceeds as follows:

1. Processor generates a virtual address,
2. MMU looks in the TLB for the referenced page.
3. If the page table entry for this page is found in the TLB, the physical address is obtained immediately.

4. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.
5. Then TLB is updated accordingly.



Use of an associative-mapped TLB

Page Faults:

- When a program generates an access request to a page that is not in the main memory, a page fault is said to have occurred. The entire page must be brought from the disk into the memory before access can proceed.
- When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the program that generated the page fault is interrupted, and control is transferred to the operating system.
- The operating system copies the requested page from the disk into the main memory and transfers the control back to the interrupted task.

Page replacement:

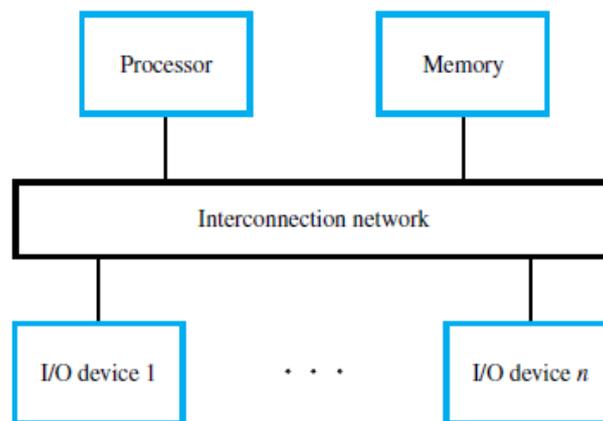
- If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the control bits in the page table entries can be used to record usage history.
- One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system periodically clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

Write operation:

- A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data.

ACCESSING I/O DEVICES:

- The components of a computer system communicate with each other through an interconnection network, as shown in Figure. The interconnection network consists of circuits needed to transfer information between the processor, the memory unit, and a number of I/O devices.
- Load and Store instructions use addressing modes to generate effective addresses that identify the desired locations. This idea of using addresses to access various locations in the memory can be extended to deal with the I/O devices as well.



A computer system

- For this purpose, each I/O device must appear to the processor as consisting of some addressable locations, just like the memory. Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory.
- These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of registers. It is customary to refer to them as I/O registers. Since the I/O devices and the memory share the same address space, this arrangement is called memory-mapped I/O. It is used in most computers.
- With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of a register in an input device, the instruction

Load R2, DATAIN

reads the data from the DATAIN register and loads them into processor register R2.

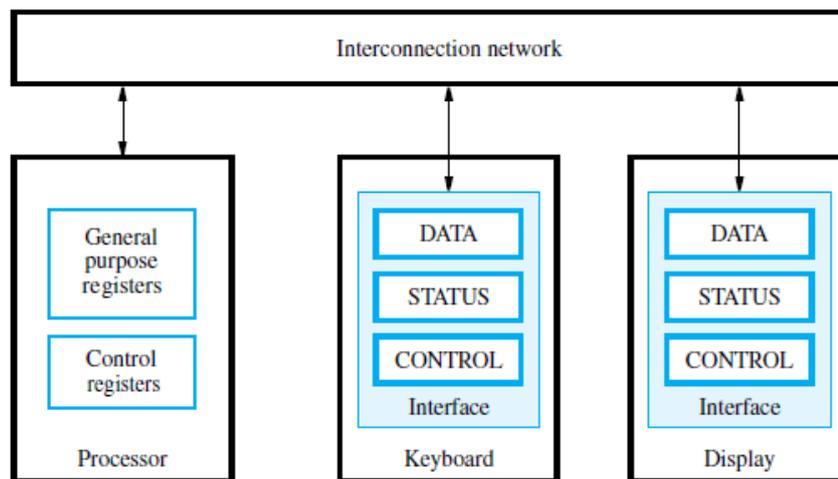
- Similarly, the instruction

Store R2, DATAOUT

sends the contents of register R2 to location DATAOUT, which is a register in an output device.

I/O Device Interface

- An I/O device is connected to the interconnection network by using a circuit, called the device interface, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor.
- One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behaviour of the device. These data, status, and control registers are accessed by program instructions as if they were memory locations. Typical transfers of information are between I/O registers and the registers in the processor.
- Figure illustrates how the keyboard and display devices are connected to the processor from the software point of view.

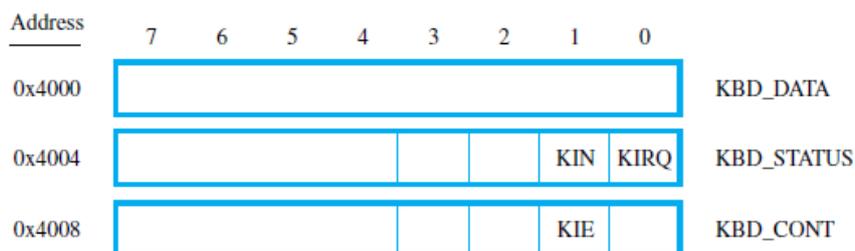


The connection for processor, keyboard, and display

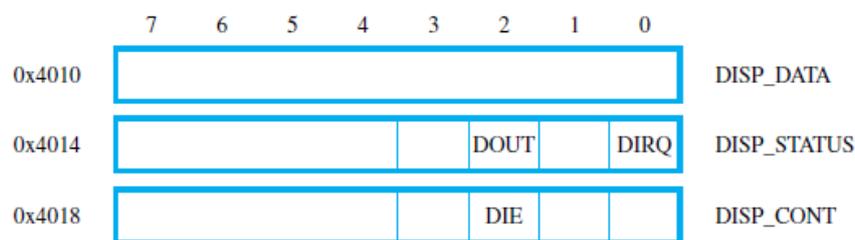
Program-Controlled I/O

- Let us begin the discussion of input/output issues by looking at two essential I/O devices for human-computer interaction—keyboard and display.
- Consider a task that reads characters typed on a keyboard, stores these data in the memory, and displays the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action. This method is known as program-controlled I/O.
- In addition to transferring each character from the keyboard into the memory, and then to the display, it is necessary to ensure that this happens at the right time. An input character must be read in response to a key being pressed.
- For output, a character must be sent to the display only when the display device is able to accept it. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second.
- The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second.
- However, this is still much slower than the speed of a processor that can execute billions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

- One solution to this problem involves a signaling protocol. On output, the processor sends the first character and then waits for a signal from the display that the next character can be sent. It then sends the second character, and so on. An input character is obtained from the keyboard in a similar way.
- The processor waits for a signal indicating that a key has been pressed and that a binary code that represents the corresponding character is available in an I/O register associated with the keyboard. Then the processor proceeds to read that code.
- The keyboard includes a circuit that responds to a key being pressed by producing the code for the corresponding character that can be used by the computer. We will assume that ASCII code is used, in which each character code occupies one byte.
- Let KBD_DATA be the address label of an 8-bit register that holds the generated character. Also, let a signal indicating that a key has been pressed be provided by setting to 1 a flip-flop called KIN, which is a part of an eight-bit status register, KBD_STATUS.
- The processor can read the status flag KIN to determine when a character code has been placed in KBD_DATA. When the processor reads the status flag to determine its state, we say that the processor polls the I/O device.
- The display includes an 8-bit register, which we will call DISP_DATA, used to receive characters from the processor. It also must be able to indicate that it is ready to receive the next character; this can be done by using a status flag called DOUT, which is one bit in a status register, DISP_STATUS.



(a) Keyboard interface



(b) Display interface

Registers in the keyboard and display interfaces

- A program is needed to perform the task of reading the characters produced by the keyboard, storing these characters in the memory, and sending them to the display. To perform I/O transfers, the processor must execute machine instructions that check the state of the status flags and transfer data between the processor and the I/O devices.

INTERRUPTS:

- Data transfer between the CPU and the peripherals is initiated by the CPU. But the CPU cannot start the transfer unless the peripheral is ready to communicate with the CPU. When a device is ready to communicate with the CPU, it generates an interrupt signal. A number of input-output devices are attached to the computer and each device is able to generate an interrupt request.
- The main job of the interrupt system is to identify the source of the interrupt. There is also a possibility that several devices will request simultaneously for CPU communication. Then, the interrupt system has to decide which device is to be serviced first.

Priority Interrupt

- A priority interrupt is a system which decides the priority at which various devices, which generates the interrupt signal at the same time, will be serviced by the CPU. The system has authority to decide which conditions are allowed to interrupt the CPU, while some other interrupt is being serviced. Generally, devices with high speed transfer such as magnetic disks are given high priority and slow devices such as keyboards are given low priority.
- When two or more devices interrupt the computer simultaneously, the computer services the device with the higher priority first.

Types of Interrupts:

Hardware Interrupts

- When the signal for the processor is from an external device or hardware then this interrupt is known as hardware interrupt. Let us consider an example: when we press any key on our keyboard to do some action, then this pressing of the key will generate an interrupt signal for the processor to perform certain action. Such an interrupt can be of two types:

Maskable Interrupt

- The hardware interrupts which can be delayed when a much high priority interrupt has occurred at the same time.

Non Maskable Interrupt

- The hardware interrupts which cannot be delayed and should be processed by the processor immediately.

Software Interrupts

- The interrupt that is caused by any internal system of the computer system is known as a software interrupt. It can also be of two types:

Normal Interrupt

- The interrupts that are caused by software instructions are called normal software interrupts.

Exception

- Unplanned interrupts which are produced during the execution of some program are called exceptions, such as division by zero.

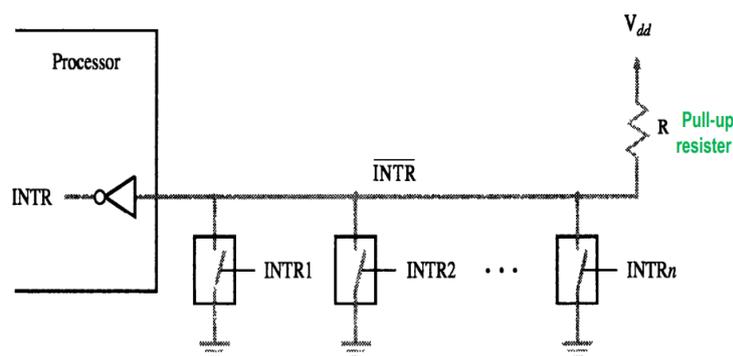
Interrupt handling mechanism:

- The basic method of interrupting the CPU is activating a special control line called INTERRUPT REQUEST which connects the interrupt source to the CPU.
- An interrupt indicator is then stored in a CPU register that can be checked by the CPU periodically. CPU does this checking at the end of every instruction cycle.
- If an interrupt is identified, the CPU executes a specific interrupt handling program.
- Each interrupt source has its own interrupt service routine. So, the CPU must determine the address of the interrupt program to be executed. If more than one interrupt requests at the same time, then priorities must be assigned and the interrupt with highest priority is selected for handling.
- The following steps are taken while handling the interrupts:
 1. IO device enables the INTERRUPT REQUEST control line.
 2. Interrupt indicator is enabled in the CPU register.
 3. The CPU identifies the source of interrupt.
 4. The CPU obtains the memory address of the required interrupt handler.
 5. The PC and other processor registers are saved in stack as subroutine call.
 6. The PC is loaded with the address of the interrupt handler. Execution proceeds until return instruction is reached. Immediately, the control is transferred back to the interrupted program.

Interrupt hardware:

- An input / output device requests an interrupt by activating a bus line called interrupt. A single interrupt request line may be used to serve n devices as shown in figure.
- All devices are connected to the line via switches to ground. To request an interrupt, a device closes its associated switch. Thus, if all interrupt-request signals $INTR_1$ to $INTR_n$ are inactive, that is, if all switches are open, the voltage on the interrupt request line will be equal to V_{dd} . This is the inactive state of the line. Since the closing of one or more switches will cause the line voltage to drop to 0, the value of $INTR$ is the logical OR of the requests from individual devices, that is,

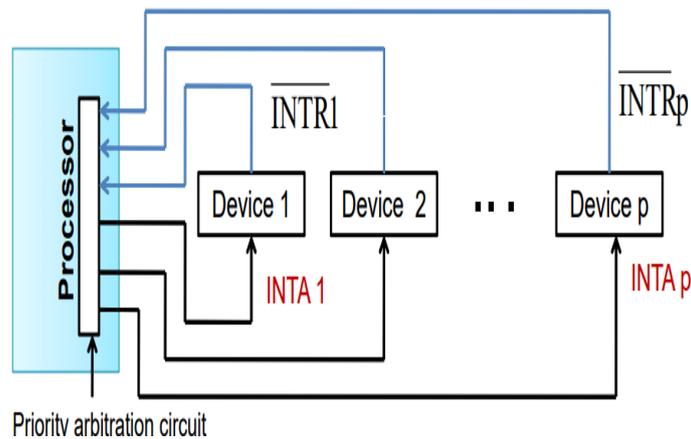
$$INTR = INTR_1 + \dots + INTR_n$$



$$INTR = INTR_1 + \dots + INTR_n$$

Interrupt priority:

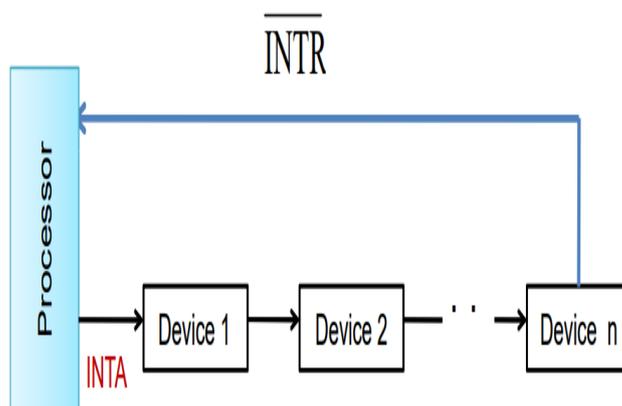
- A multiple-level priority organization means that during execution of an interrupt-service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority.
- To implement this scheme, we can assign a priority level to the processor that can be changed under program control. The priority level of the processor is the priority of the program that is currently being executed. The processor accepts interrupts only from devices that have priorities higher than its own.
- A multiple-priority scheme can be implemented easily by using separate interrupt-request and interrupt-acknowledge lines for each device, as shown in figure.
- Each of the interrupt-request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor.
- A request is accepted only if it has a higher priority level than that currently assigned to the processor.



Implementation of priority groups

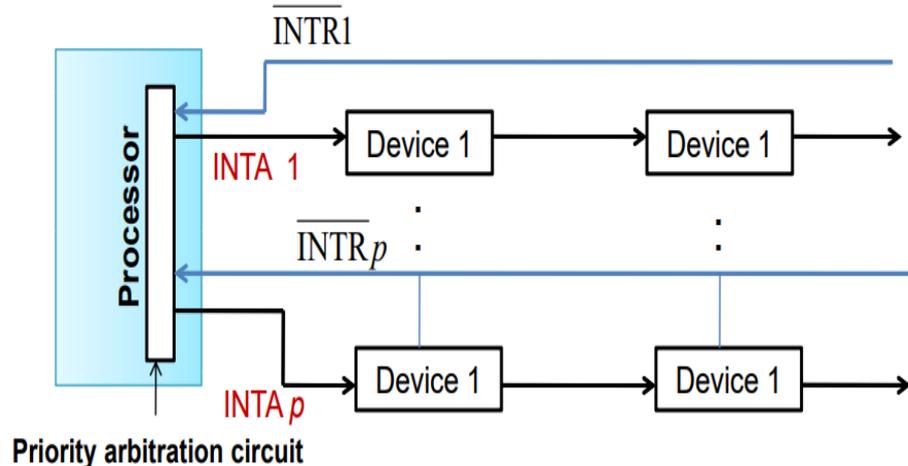
Simultaneous Requests:-

- Let us now consider the problem of simultaneous arrivals of interrupt requests from two or more devices. The processor must have some means of deciding which requests to service first. Using a priority scheme such as that of figure, the solution is straightforward. The processor simply accepts the requests having the highest priority.
- Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a daisy chain, as shown in figure.
- The interrupt-request line, INTR, The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices.



Daisy chain interrupt

- When several devices raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1.
- Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.
- The scheme in figure 2 requires considerably fewer wires than the individual connections in figure 2. The main advantage of the scheme in figure 1 is that it allows the processor to accept interrupt requests from some devices but not from others, depending upon their priorities. The two schemes may be combined to produce the more general structure in below figure.
- Devices are organized in groups, and each group is connected at a different priority level. Within a group, devices are connected in a daisy chain. This organization is used in many computer systems.



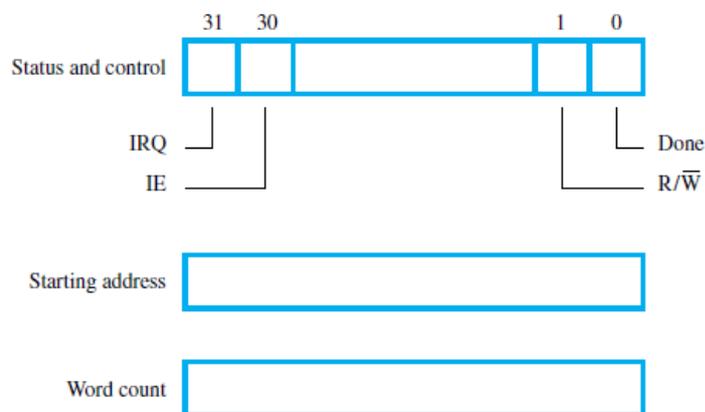
Arrangement of priority groups

DIRECT MEMORY ACCESS:

- Blocks of data are often transferred between the main memory and I/O devices such as disks. This section discusses a technique for controlling such transfers without frequent, program-controlled intervention by the processor.
- Data are transferred from an I/O device to the memory by first reading them from the I/O device using an instruction such as Load R2, DATAIN which loads the data into a processor register. Then, the data read are stored into a memory location. The reverse process takes place for transferring data from the memory to an I/O device.
- An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready, either by polling its status register or by waiting for an interrupt request.
- In either case, considerable overhead is incurred, because several program instructions must be executed involving many memory accesses for each data word transferred. When transferring a block of data, instructions are needed to increment the memory address and keep track of the word count.
- The use of interrupts involves operating system routines which incur additional overhead to save and restore processor registers, the program counter, and other state information.
- An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks.
- A special control unit is provided to manage the transfer, without continuous intervention by the processor. This approach is called direct memory access, or DMA. The unit that controls DMA

transfers is referred to as a DMA controller. It may be part of the I/O device interface, or it may be a separate unit shared by a number of I/O devices.

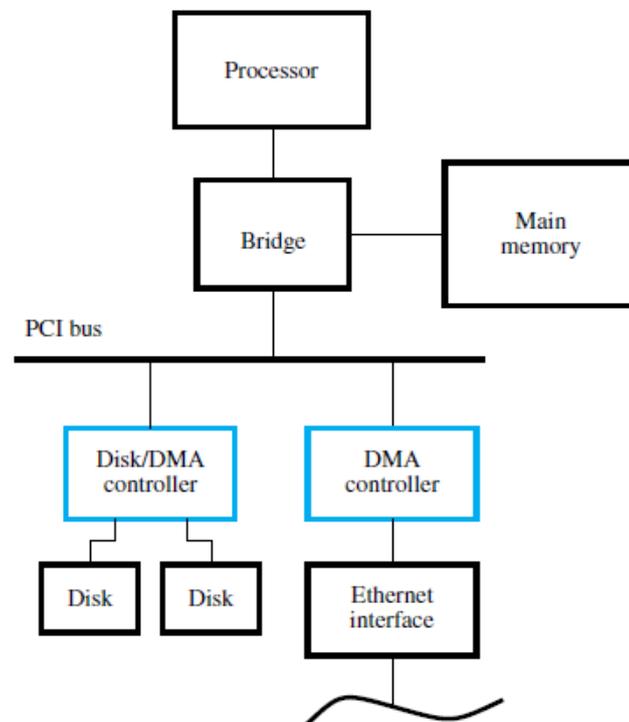
- The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory.
- For each word transferred, it provides the memory address and generates all the control signals needed. It increments the memory address for successive words and keeps track of the number of transfers.
- Although a DMA controller transfers data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an operating system routine.
- To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, the number of words in the block, and the direction of the transfer.
- The DMA controller then proceeds to perform the requested operation. When the entire block has been transferred, it informs the processor by raising an interrupt.
- Figure shows an example of the DMA controller registers that are accessed by the processor to initiate data transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The R/W bit determines the direction of the transfer.
- When this bit is set to 1 by a program instruction, the controller performs a Read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a Write operation. Additional information is also transferred as may be required by the I/O device.
- For example, in the case of a disk, the processor provides the disk controller with information to identify where the data is located on the disk.



Typical registers in a DMA controller

- When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE.
- When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt.
- Figure shows how DMA controllers may be used in a computer system. One DMA controller connects a high-speed Ethernet to the computer's I/O bus.
- The disk controller, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller.
- The registers needed to store the memory address, the word count, and so on, are duplicated, so that one set can be used with each disk.

- To start a DMA transfer of a block of data from the main memory to one of the disks, an OS routine writes the address and word count information into the registers of the disk controller. The DMA controller proceeds independently to implement the specified operation.
- When the transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit.
- At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. The status register may also be used to record other information, such as whether the transfer took place correctly or errors occurred.



Use of DMA controllers in a computer system

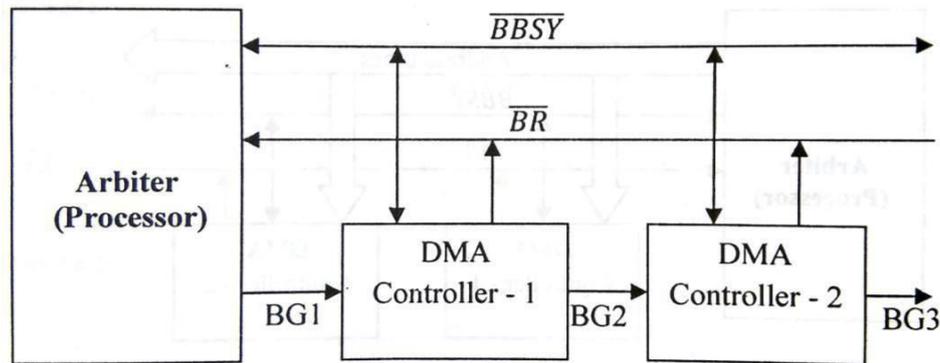
Bus Arbitration

- The device that is allowed to initiate data transfers on the bus at any given time is called bus master.
- Bus Arbitration is the process by which the next device to become the bus master is selected and bus master ship is transferred to it. The selection bus master is usually done on the priority basis. Approaches to bus Arbitration:
 - Centralized Bus Arbitration
 - Distributed bus arbitration
- In centralized bus arbitration, a single bus arbiter performs the required arbitration. The bus arbiter may be the processor or a separate controller connected to the bus.
- There are different arbitration scheme that use centralized bus arbitration approach. These schemes are:
 - Daisy chaining
 - Polling method
 - Independent request

Daisy chaining:

- Daisy chain method uses three lines.
- **Bus Request Line (BR):** This is a wired-OR line. The controller only knows that a request has been made by a device, but doesn't know which device made the request.

- **Bus Grant Line (BG):** Bus grant signal is propagated through all the devices through this line.
- **Bus Busy (BBSY):** The current bus master indicates to all devices that it is using the bus by activating this open-collector line called Bus Busy.



- In daisy chaining method, all masters use the same bus request line for requesting the bus. In response to the bus request, the controller sends a bus grant signal BG if the bus is free. This grant signal is propagated to the masters until it reaches the master which requests the bus. This master blocks the bus grant signal, activates the bus busy line and gets the control of the bus. If more than one device makes a request at the same time, then the device that is closer to the arbiter will get the bus. This is known as daisy chaining.

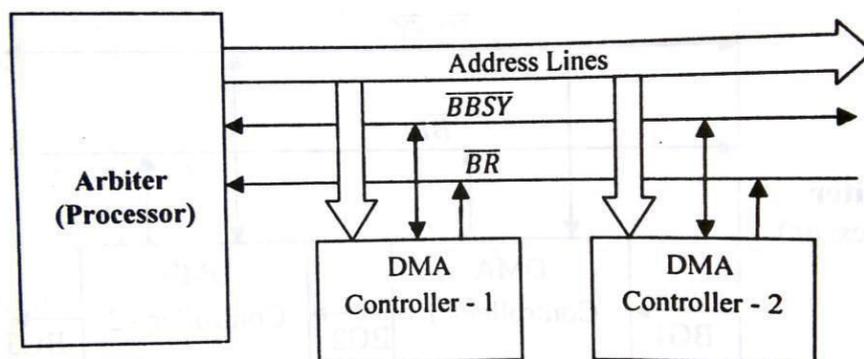
Advantages:

- It is a simple and cheaper method.
- It requires the least number of lines and this number is independent of the number of masters in the system.

Disadvantages:

- The propagation delay of bus grant signal is proportional to the number of masters in the system. This makes arbitration time slow and hence limits the number of masters in the system.
- The priority of the master is fixed by its physical location.
- Failure of any one master causes the whole system to fail.

Polling Method:



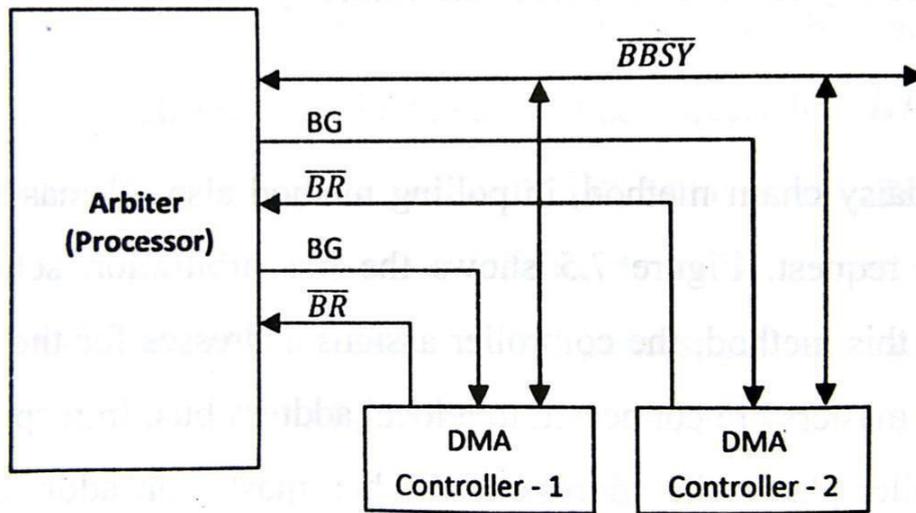
- Similar to daisy chain method, in polling method also all masters use the same line for bus request. In this method the controller assigns addresses for the masters in the system. All the masters are connected to a local address bus.
- In response to bus request, the controller places the address of the bus master on the address bus. The requesting master recognizes its address, activates the busy line and gets the control of the bus.

Advantages:

- The priority can be changed by altering the polling sequence stored in the controller.
- If the one module fails entire system does not fail.

Independent Request Method:

- In this scheme each master has a separate pair of bus request and bus grant lines and each pair has a priority assigned to it.
- The built in priority decoder within the controller selects the highest priority request and asserts the corresponding grant signal.



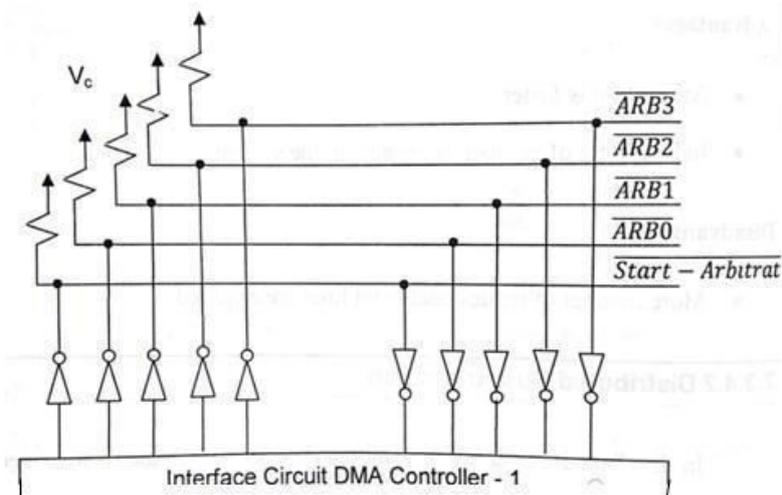
Advantage:

- Due to separate pairs of bus request and bus grant signals, arbitration is fast and is independent of the number of masters in the system.

Disadvantage:

- It requires more bus request and grant signals.

Distributed bus arbitration:



- In distributed bus arbitration, all devices participate in the selection of the next bus master. In this scheme each device is assigned with a 4-bit identification number.

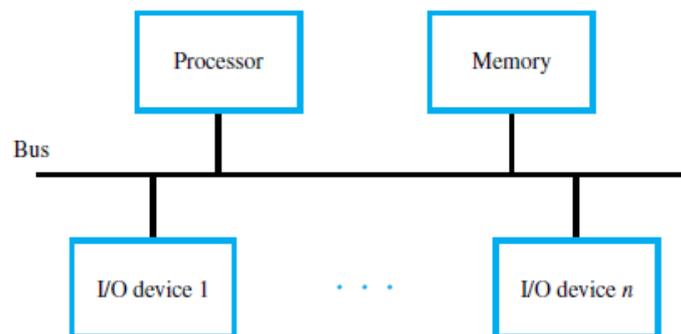
- When a device requests for the control of the bus, it enables the $\overline{start - arbitration}$ signal and places its 4-bit ID on arbitration lines $\overline{ARB0}$ through $\overline{ARB3}$.
- These arbitration lines are open collectors. Therefore more than one device can place their ID to indicate that they also need the control of the bus.
- In this scheme the device which has the highest ID number has the highest priority.

Advantages:

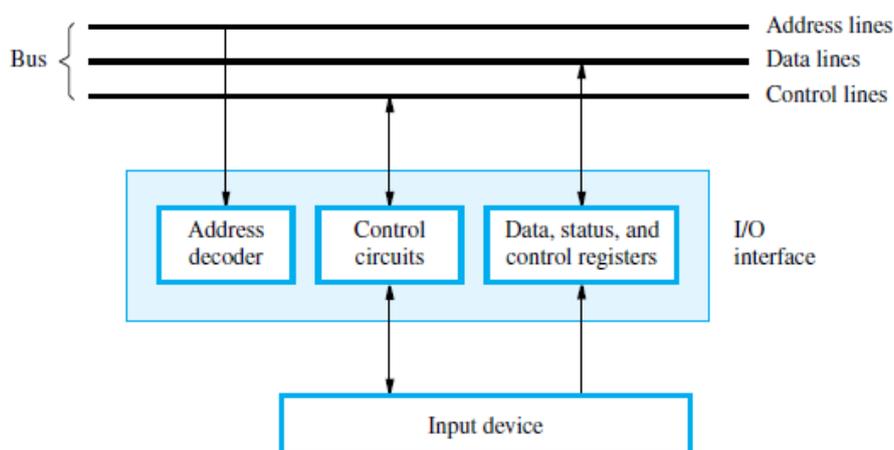
- Decentralized arbitration provides higher reliability.
- The operation of the bus is not dependent on any single device.

BUS STRUCTURE:

- The bus shown in Figure is a simple structure that implements the interconnection Network. Only one source/destination pair of units can use this bus to transfer data at any one time.
- The bus consists of three sets of lines used to carry address, data, and control signals. I/O device interfaces are connected to these lines, as shown in Figure for an input device.
- Each I/O device is assigned a unique set of addresses for the registers in its interface. When the processor places a particular address on the address lines, it is examined by the address decoders of all devices on the bus.
- The device that recognizes this address responds to the commands issued on the control lines. The processor uses the control lines to request either a Read or a Write operation, and the requested data are transferred over the data lines.



A single-bus structure



I/O interface for an input device

- When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O. Any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if the input device in Figure is a keyboard and if DATAIN is its data register, the instruction Load R2, DATAIN reads the data from DATAIN and stores them into processor register R2.
- Similarly, the instruction Store R2, DATAOUT sends the contents of register R2 to location DATAOUT, which may be the data register of a display device interface.
- The status and control registers contain information relevant to the operation of the I/O device. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

BUS OPERATION:

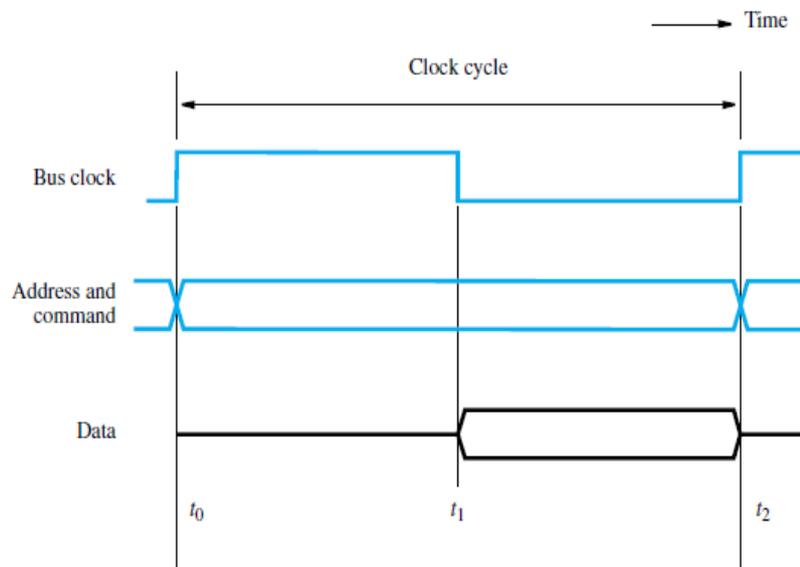
- A bus requires a set of rules, often called a bus protocol, that govern how the bus is used by various devices. The bus protocol determines when a device may place information on the bus, when it may load the data on the bus into one of its registers, and so on. These rules are implemented by control signals that indicate what and when actions are to be taken.
- One control line, usually labelled R/\bar{W} , specifies whether a Read or a Write operation is to be performed.
- As the label suggests, it specifies Read when set to 1 and Write when set to 0. When several data sizes are possible, such as byte, half word, or word, the required size is indicated by other control lines. The bus control lines also carry timing information.
- They specify the times at which the processor and the I/O devices may place data on or receive data from the data lines. A variety of schemes have been devised for the timing of data transfers over a bus. These can be broadly classified as either synchronous or asynchronous schemes.
- In any data transfer operation, one device plays the role of a master. This is the device that initiates data transfers by issuing Read or Write commands on the bus. The device addressed by the master is referred to as a slave.

Synchronous Bus

- On a synchronous bus, all devices derive timing information from a control line called the bus clock. The signal on this line has two phases: a high level followed by a low level. The two phases constitute a clock cycle.
- The first half of the cycle between the low-to-high and high-to-low transitions is often referred to as a clock pulse. The address and data lines in Figure are shown as if they are carrying both high and low signal levels at the same time. This is a common convention for indicating that some lines are high and some low, depending on the particular address or data values being transmitted.
- The crossing points indicate the times at which these patterns change. A signal line at a level half-way between the low and high signal levels indicates periods during which the signal is unreliable, and must be ignored by all devices.
- Let us consider the sequence of signal events during an input (Read) operation. At time t_0 , the master places the device address on the address lines and sends a command on the control lines indicating a Read operation. The command may also specify the length of the operand to be read. Information travels over the bus at a speed determined by its physical and electrical characteristics.
- The clock pulse width, $t_1 - t_0$, must be longer than the maximum propagation delay over the bus. Also, it must be long enough to allow all devices to decode the address and control signals, so

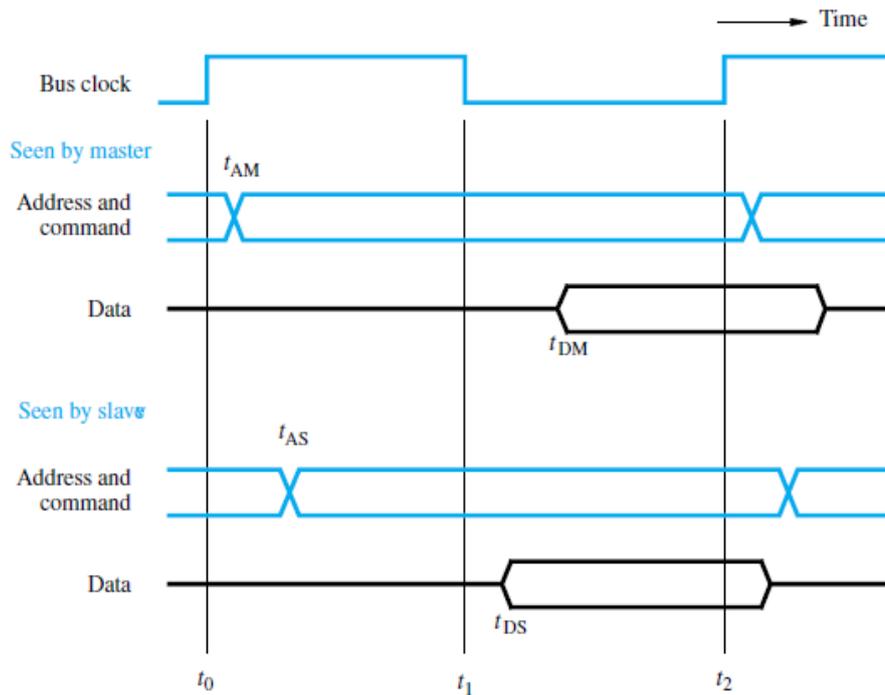
that the addressed device (the slave) can respond at time t_1 by placing the requested input data on the data lines.

- At the end of the clock cycle, at time t_2 , the master loads the data on the data lines into one of its registers.
- To be loaded correctly into a register, data must be available for a period greater than the setup time of the register. Hence, the period $t_2 - t_1$ must be greater than the maximum propagation time on the bus plus the setup time of the master's register.



a. Timing of an input transfer on a synchronous bus

- A similar procedure is followed for a Write operation. The master places the output data on the data lines when it transmits the address and command information. At time t_2 , the addressed device loads the data into its data register.
- Figure b gives a more realistic picture of what actually happens. It shows two views of each signal, except the clock. Because signals take time to travel from one device to another, a given signal transition is seen by different devices at different times.
- The top view shows the signals as seen by the master and the bottom view as seen by the slave. We assume that the clock changes are seen at the same time by all devices connected to the bus. System designers spend considerable effort to ensure that the clock signal satisfies this requirement.
- The master sends the address and command signals on the rising edge of the clock at the beginning of the clock cycle (at t_0). However, these signals do not actually appear on the bus until t_{AM} , largely due to the delay in the electronic circuit output from the master to the bus lines. A short while later, at t_{AS} , the signals reach the slave.
- The slave decodes the address, and at t_1 sends the requested data. Here again, the data signals do not appear on the bus until t_{DS} . They travel toward the master and arrive at t_{DM} .
- At t_2 , the master loads the data into its register. Hence the period $t_2 - t_{DM}$ must be greater than the setup time of that register. The data must continue to be valid after t_2 for a period equal to the hold time requirement of the register.

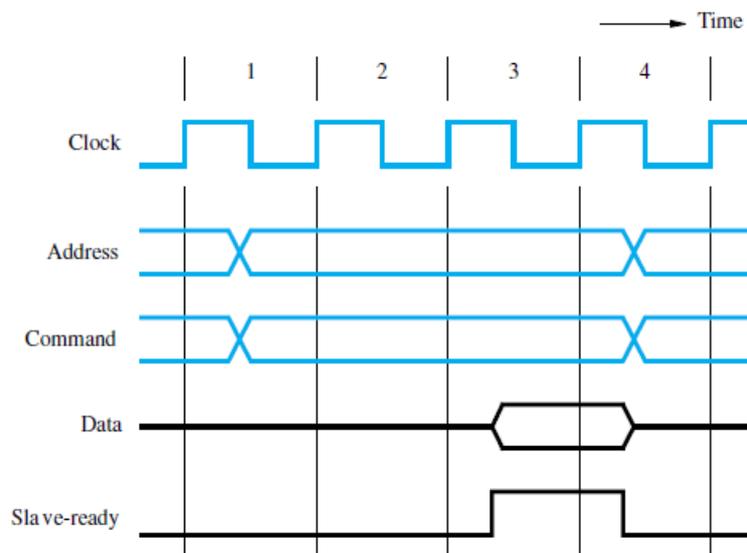


b. A detailed timing diagram for the input transfer of Figure a

Multiple-Cycle Data Transfer

- The scheme described above results in a simple design for the device interface. However, it has some limitations. Because a transfer has to be completed within one clock cycle, the clock period, $t_2 - t_0$, must be chosen to accommodate the longest delays on the bus and the slowest device interface. This forces all devices to operate at the speed of the slowest device.
- Also, the processor has no way of determining whether the addressed device has actually responded. At t_2 , it simply assumes that the input data are available on the data lines in a Read operation, or that the output data have been received by the I/O device in a Write operation.
- If, because of a malfunction, a device does not operate correctly, the error will not be detected.
- To overcome these limitations, most buses incorporate control signals that represent a response from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data transfer operation. They also make it possible to adjust the duration of the data transfer period to match the response speeds of different devices. This is often accomplished by allowing a complete data transfer operation to span several clock cycles. Then, the number of clock cycles involved can vary from one device to another.
- An example of this approach is shown in Figure. During clock cycle 1, the master sends address and command information on the bus, requesting a Read operation. The slave receives this information and decodes it. It begins to access the requested data on the active edge of the clock at the beginning of clock cycle 2.
- We have assumed that due to the delay involved in getting the data, the slave cannot respond immediately. The data become ready and are placed on the bus during clock cycle 3.

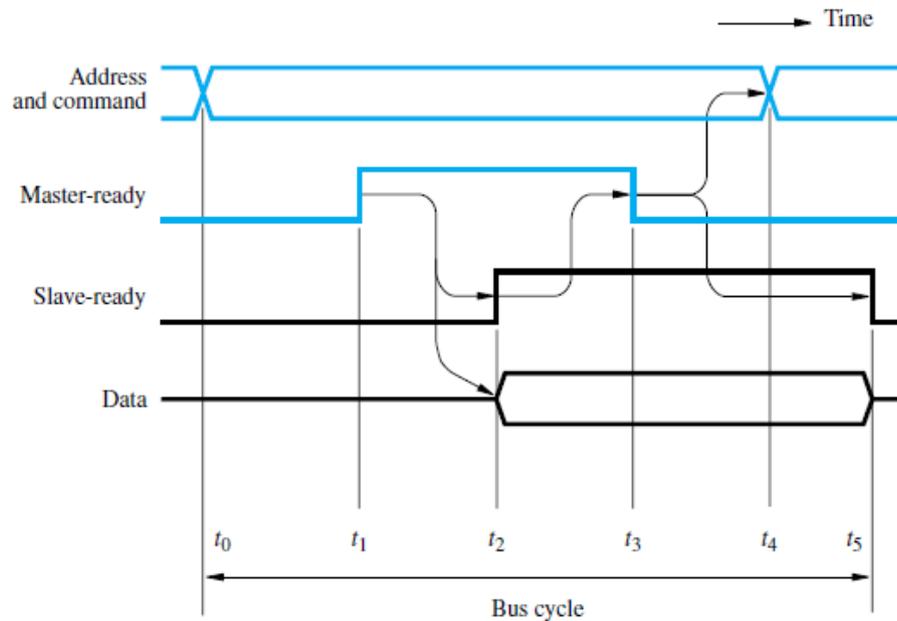
- The slave asserts a control signal called Slave-ready at the same time. The master, which has been waiting for this signal, loads the data into its register at the end of the clock cycle. The slave removes its data signals from the bus and returns its Slave-ready signal to the low level at the end of cycle 3.
- The bus transfer operation is now complete, and the master may send new address and command signals to start a new transfer in clock cycle 4.
- The Slave-ready signal is an acknowledgment from the slave to the master, confirming that the requested data have been placed on the bus. It also allows the duration of a bus transfer to change from one device to another.



An input transfer using multiple clock cycles

Asynchronous Bus

- An alternative scheme for controlling data transfers on a bus is based on the use of a handshake protocol between the master and the slave. A handshake is an exchange of command and response signals between the master and the slave.
- A control line called Master-ready is asserted by the master to indicate that it is ready to start a data transfer. The Slave responds by asserting Slave-ready.
- A data transfer controlled by a handshake protocol proceeds as follows. The master places the address and command information on the bus. Then it indicates to all devices that it has done so by activating the Master-ready line. This causes all devices to decode the address. The selected slave performs the required operation and informs the processor that it has done so by activating the Slave-ready line.
- The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a Read operation, it also loads the data into one of its registers.

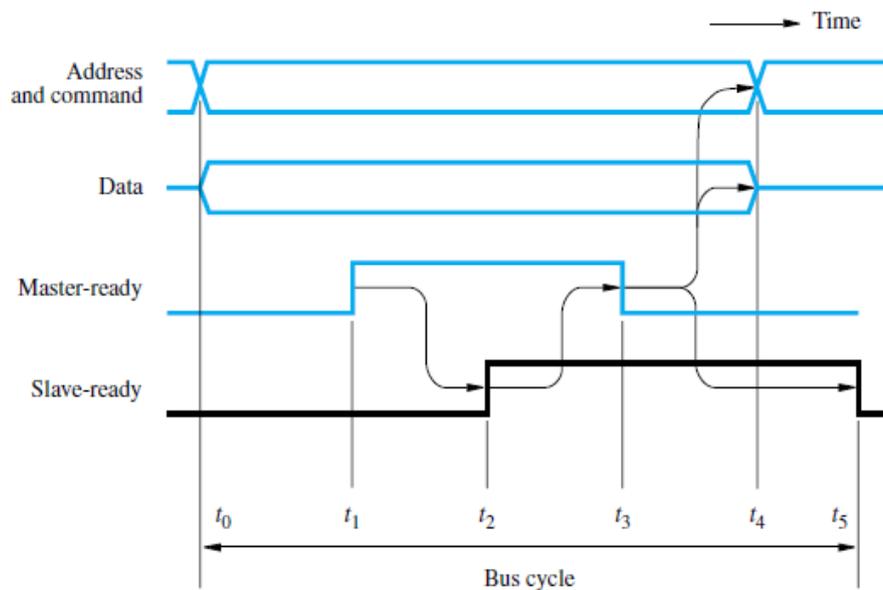


c. Handshake control of data transfer during an input operation

➤ An example of the timing of an input data transfer using the handshake protocol is given in Figure, which depicts the following sequence of events:

- t_0 — The master places the address and command information on the bus, and all devices on the bus decode this information.
- t_1 — The master sets the Master-ready line to 1 to inform the devices that the address and command information is ready. The delay $t_1 - t_0$ is intended to allow for any skew that may occur on the bus. Skew occurs when two signals transmitted simultaneously from one source arrive at the destination at different times. This happens because different lines of the bus may have different propagation speeds. Thus, to guarantee that the Master-ready signal does not arrive at any device ahead of the address and command information, the delay $t_1 - t_0$ should be longer than the maximum possible bus skew. Sufficient time should be allowed for the device interface circuitry to decode the address. The delay needed can be included in the period $t_1 - t_0$.
- t_2 —The selected slave, having decoded the address and command information, performs the required input operation by placing its data on the data lines. At the same time, it sets the Slave-ready signal to 1. If extra delays are introduced by the interface circuitry before it places the data on the bus, the slave must delay the Slave-ready signal accordingly. The period $t_2 - t_1$ depends on the distance between the master and the slave and on the delays introduced by the slave's circuitry.
- t_3 —The Slave-ready signal arrives at the master, indicating that the input data are available on the bus. The master must allow for bus skew. It must also allow for the setup time needed by its register. After a delay equivalent to the maximum bus skew and the minimum setup time, the master loads the data into its register. Then, it drops the Master-ready signal, indicating that it has received the data.

- t_4 —The master removes the address and command information from the bus. The delay between t_3 and t_4 is again intended to allow for bus skew. Erroneous addressing may take place if the address, as seen by some device on the bus, starts to change while the Master-ready signal is still equal to 1.
 - t_5 —When the device interface receives the 1-to-0 transition of the Master-ready signal, it removes the data and the Slave-ready signal from the bus. This completes the input transfer.
- The timing for an output operation, illustrated in Figure d, is essentially the same as for an input operation.
- In this case, the master places the output data on the data lines at the same time that it transmits the address and command information. The selected slave loads the data into its data register when it receives the Master-ready signal and indicates that it has done so by setting the Slave-ready signal to 1. The remainder of the cycle is similar to the input operation.
- The handshake signals in Figures c and d are said to be fully interlocked, because a change in one signal is always in response to a change in the other. Hence, this scheme is known as a full handshake. It provides the highest degree of flexibility and reliability.



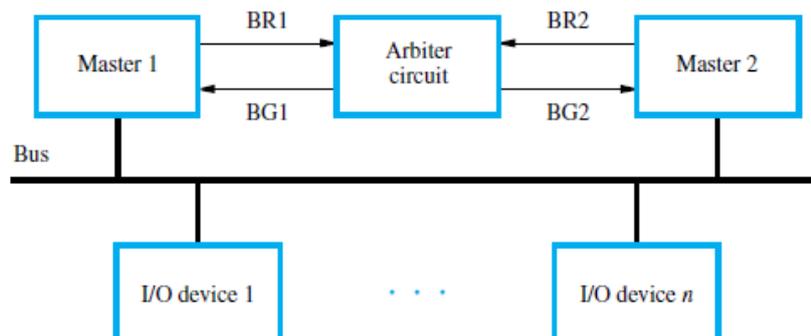
d. Handshake control of data transfer during an output operation

- The main advantage of the asynchronous bus is that the handshake protocol eliminates the need for distribution of a single clock signal whose edges should be seen by all devices at about the same time. This simplifies timing design.
- Delays, whether introduced by the interface circuits or by propagation over the bus wires, are readily accommodated. These delays are likely to differ from one device to another, but the timing of data transfers adjusts automatically.
- For a synchronous bus, clock circuitry must be designed carefully to ensure proper timing, and delays must be kept within strict bounds.

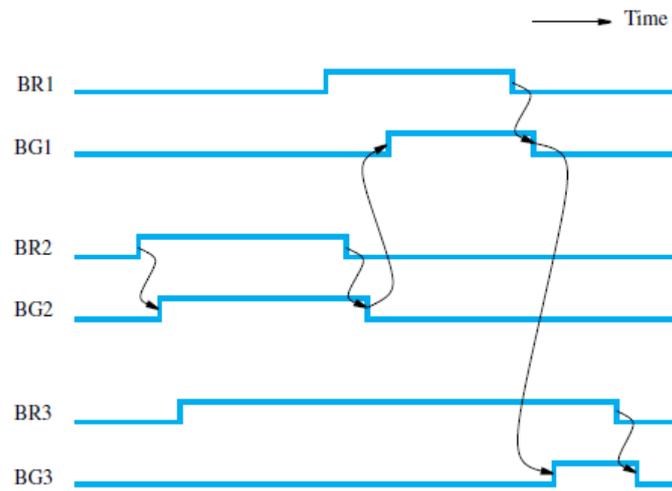
- The rate of data transfer on an asynchronous bus controlled by the handshake protocol is limited by the fact that each transfer involves two round-trip delays.
- On synchronous buses, the clock period need only accommodate one round trip delay. Hence, faster transfer rates can be achieved.

ARBITRATION

- There are occasions when two or more entities contend for the use of a single resource in a computer system. For example, two devices may need to access a given slave at the same time. In such cases, it is necessary to decide which device will access the slave first.
- The decision is usually made in an arbitration process performed by an arbiter circuit. The arbitration process starts by each device sending a request to use the shared resource. The arbiter associates priorities with individual requests.
- If it receives two requests at the same time, it grants the use of the slave to the device having the higher priority first.
- To illustrate the arbitration process, we consider the case where a single bus is the shared resource. The device that initiates data transfer requests on the bus is the bus master.
- It is possible that several devices in a computer system need to be bus masters to transfer data. For example, an I/O device needs to be a bus master to transfer data directly to or from the computer's memory. Since the bus is a single shared facility, it is essential to provide orderly access to it by the bus masters.
- A device that wishes to use the bus sends a request to the arbiter. When multiple requests arrive at the same time, the arbiter selects one request and grants the bus to the corresponding device. For some devices, a delay in gaining access to the bus may lead to an error. Such devices must be given high priority.
- If there is no particular urgency among requests, the arbiter may grant the bus using a simple round-robin scheme.
- Figure illustrates an arrangement for bus arbitration involving two masters. There are two Bus-request lines, BR1 and BR2, and two Bus-grant lines, BG1 and BG2, connecting the arbiter to the masters. A master requests use of the bus by activating its Bus-request line.
- If a single Bus-request is activated, the arbiter activates the corresponding Bus-grant. This indicates to the selected master that it may now use the bus for transferring data. When the transfer is completed, that master deactivates its Bus-request, and the arbiter deactivates its Bus-grant.



Bus arbitration



Granting use of the bus based on priorities

- Figure illustrates a possible sequence of events for the case of three masters. Assume that master 1 has the highest priority, followed by the others in increasing numerical order. Master 2 sends a request to use the bus first. Since there are no other requests, the arbiter grants the bus to this master by asserting BG2.
- When master 2 completes its data transfer operation, it releases the bus by deactivating BR2. By that time, both masters 1 and 3 have activated their request lines. Since device 1 has a higher priority, the arbiter activates BG1 after it deactivates BG2, thus granting the bus to master 1.
- Later, when master 1 releases the bus by deactivating BR1, the arbiter deactivates BG1 and activates BG3 to grant the bus to master 3. Note that the bus is granted to master 1 before master 3 even though master 3 activated its request line before master 1.

INTERFACE CIRCUITS:

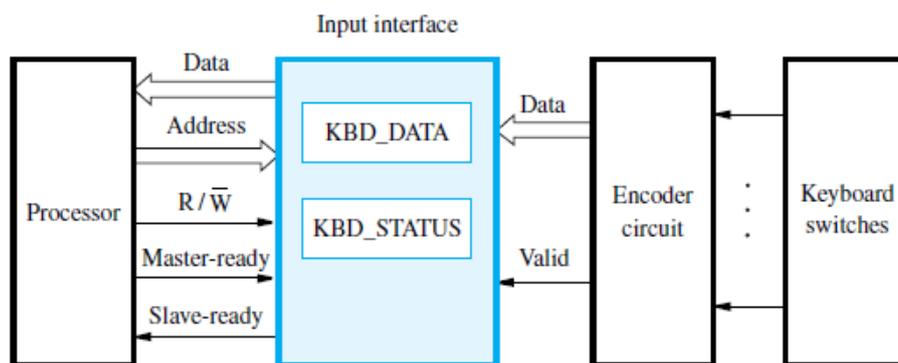
- The I/O interface of a device consists of the circuitry needed to connect that device to the bus. On one side of the interface are the bus lines for address, data, and control. On the other side are the connections needed to transfer data between the interface and the I/O device. This side is called a *port*, and it can be either a parallel or a serial port.
- A parallel port transfers multiple bits of data simultaneously to or from the device. A serial port sends and receives data one bit at a time. Communication with the processor is the same for both formats; the conversion from a parallel to a serial format and vice versa takes place inside the interface circuit.
- Before we present specific circuit examples, let us recall the functions of an I/O interface. I/O interface does the following:
 - Provides a register for temporary storage of data
 - Includes a status register containing status information that can be accessed by the processor
 - Includes a control register that holds the information governing the behavior of the interface
 - Contains address-decoding circuitry to determine when it is being addressed by the processor
 - Generates the required timing signals
 - Performs any format conversion that may be necessary to transfer data between the processor and the I/O device, such as parallel-to-serial conversion in the case of a serial port.

Parallel Interface

- First, we describe an interface circuit for an 8-bit input port that can be used for connecting a simple input device, such as a keyboard. Then, we describe an interface circuit for an 8-bit output port, which can be used with an output device such as a display.

Input Interface

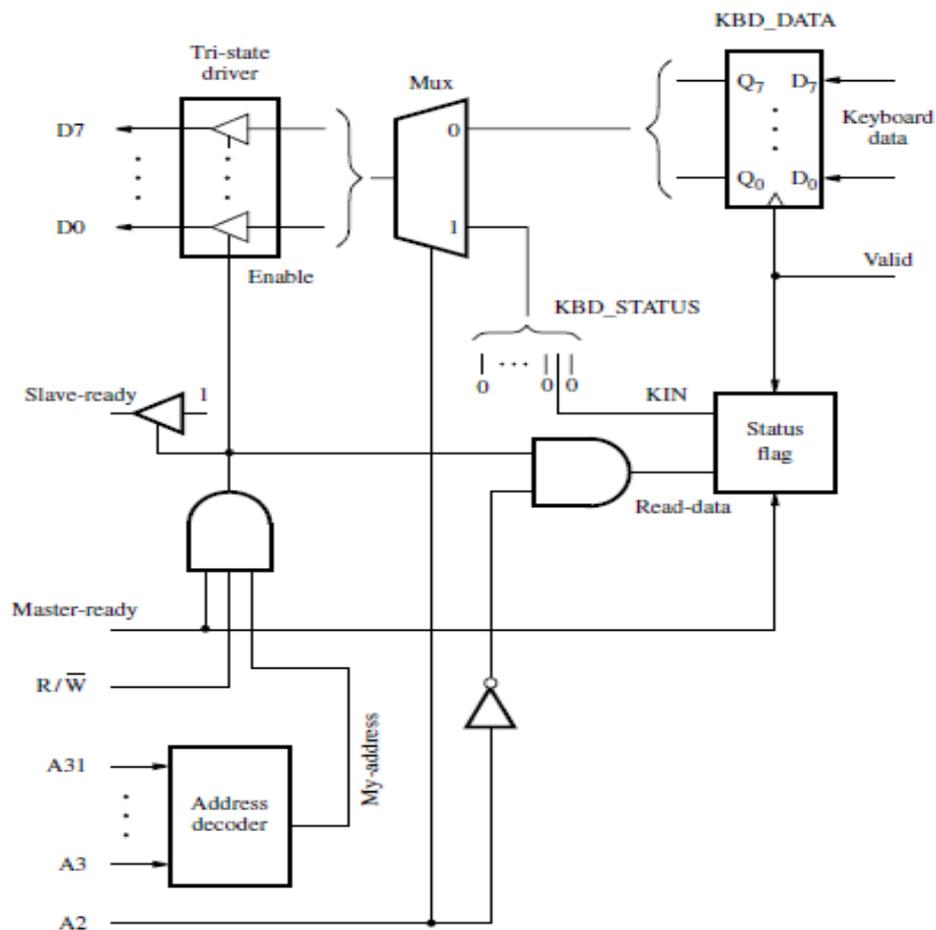
- Figure shows a circuit that can be used to connect a keyboard to a processor. Assume that interrupts are not used, so there is no need for a control register. There are only two registers: a data register, KBD_DATA, and a status register, KBD_STATUS. The latter contains the keyboard status flag, KIN.
- A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is detected by an encoder circuit that generates the ASCII code for the corresponding character.
- A difficulty with such mechanical pushbutton switches is that the contacts bounce when a key is pressed, resulting in the electrical connection being made then broken several times before the switch settles in the closed position. Although bouncing may last only one or two milliseconds, this is long enough for the computer to erroneously interpret a single pressing of a key as the key being pressed and released several times.
- The effect of bouncing can be eliminated using a simple debouncing circuit, which could be part of the keyboard hardware or may be incorporated in the encoder circuit. Alternatively, switch bouncing can be dealt with in software.
- The software detects that a key has been pressed when it observes that the keyboard status flag, KIN, has been set to 1. The I/O routine can then introduce sufficient delay before reading the contents of the input buffer, KBD_DATA, to ensure that bouncing has subsided.
- When debouncing is implemented in hardware, the I/O routine can read the input character as soon as it detects that KIN is equal to 1.



Keyboard to processor connection

- The output of the encoder in Figure consists of one byte of data representing the encoded character and one control signal called Valid. When a key is pressed, the Valid signal changes from 0 to 1, causing the ASCII code of the corresponding character to be loaded into the KBD_DATA register and the status flag KIN to be set to 1.
- The status flag is cleared to 0 when the processor reads the contents of the KBD_DATA register. The interface circuit is shown connected to an asynchronous bus on which transfers are controlled by the handshake signals Master-ready and Slave-ready. The bus has one other control line, R/\bar{W} , which indicates a Read operation when equal to 1.

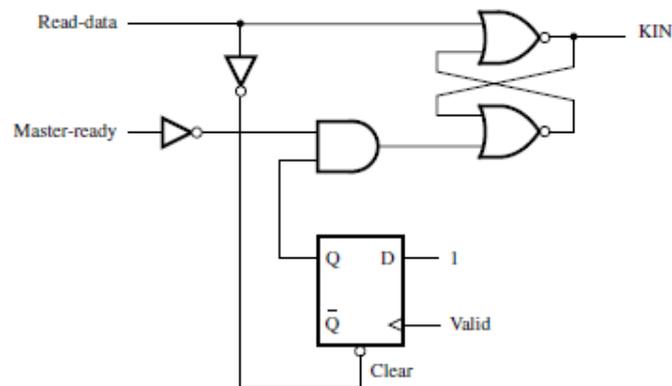
- Figure shows a possible circuit for the input interface. There are two addressable locations in this interface, KBD_DATA and KBD_STATUS. They occupy adjacent word locations in the address space.
- Only one bit, b_1 , in the status register actually contains useful information. This is the keyboard status flag, KIN. When the status register is read by the processor, all other bit locations appear as containing zeros.
- When the processor requests a Read operation, it places the address of the appropriate register on the address lines of the bus. The address decoder in the interface circuit examines bits A31–3, and asserts its output, My-address, when one of the two registers KBD_DATA or KBD_STATUS is being addressed.
- Bit A2 determines which of the two registers is involved. Hence, a multiplexer is used to select the register to be connected to the bus based on address bit A2. The two least-significant address bits, A1 and A0, are not used, because we have assumed that all addresses are word-aligned.
- The output of the multiplexer is connected to the data lines of the bus through a set of tri-state gates. The interface circuit turns the tri-state gates on only when the three signals Master-ready, My_address, and R/W are all equal to 1, indicating a Read operation. The Slave-ready signal is asserted at the same time, to inform the processor that the requested data or status information has been placed on the the data lines. When address bit A2 is equal to 0, Read-data is also asserted. This signal is used to reset the KIN flag.



An input interface circuit

- A possible implementation of the status flag circuit is given in Figure. The KIN flag is the output of a NOR latch connected as shown. A flip-flop is set to 1 by the rising edge on the Valid signal

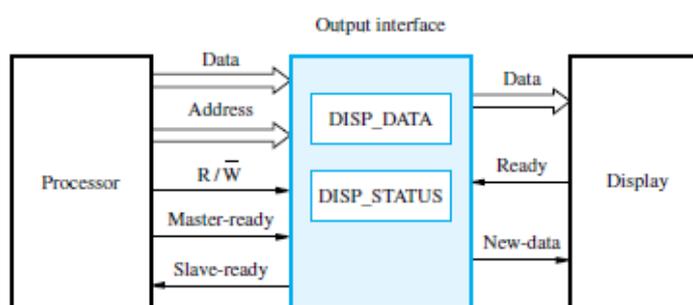
line. This event changes the state of the NOR latch to set KIN to 1, but only when Master-ready is low. The reason for this additional condition is to ensure that KIN does not change state while being read by the processor. Both the flip-flop and the latch are reset to 0 when Read-data becomes equal to 1, indicating that KBD_DATA is being read.



Circuit for the status flag block

Output Interface

- Let us now consider the output interface shown in Figure, which can be used to connect an output device such as a display. We have assumed that the display uses two handshake signals, New-data and Ready, in a manner similar to the handshake between the bus signals Master-ready and Slave-ready.
- When the display is ready to accept a character, it asserts its Ready signal, which causes the DOUT flag in the DISP_STATUS register to be set to 1.
- When the I/O routine checks DOUT and finds it equal to 1, it sends a character to DISP_DATA. This clears the DOUT flag to 0 and sets the New-data signal to 1. In response, the display returns Ready to 0 and accepts and displays the character in DISP_DATA.
- When it is ready to receive another character, it asserts Ready again, and the cycle repeats.



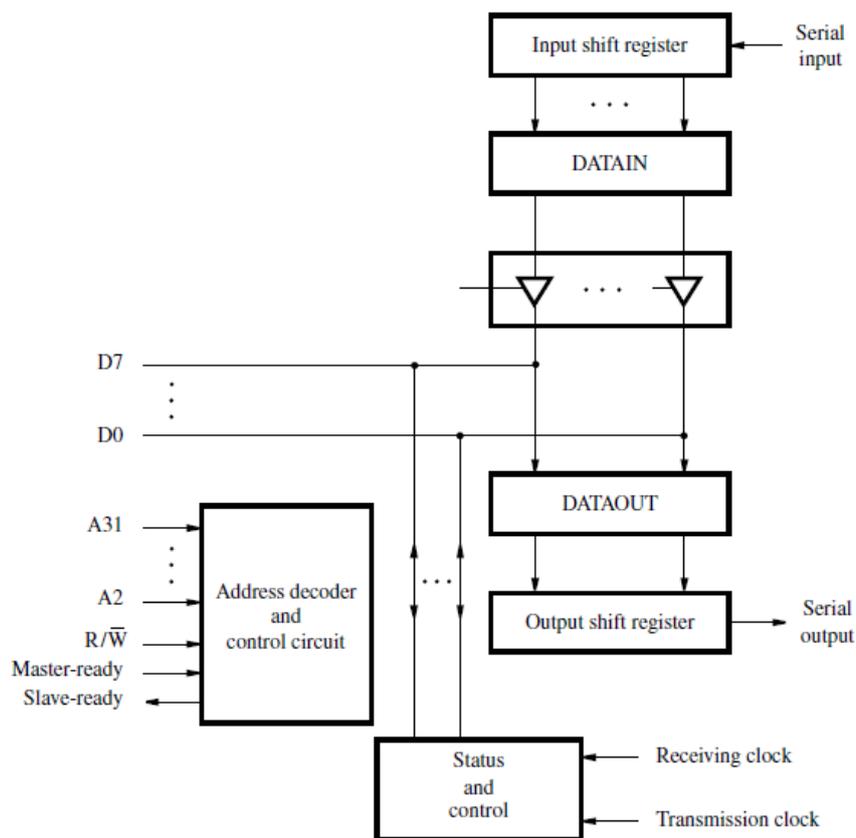
Display to processor connection

Serial Interface

- A serial interface is used to connect the processor to I/O devices that transmit data one bit at a time. Data are transferred in a bit-serial fashion on the device side and in a bit-parallel fashion on the processor side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability.
- A block diagram of a typical serial interface is shown in Figure. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the

DATAOUT register are transferred to the output shift register, from which the bits are shifted out and sent to the I/O device.

- Two status flags, which we will refer to as SIN and SOUT, are maintained by the Status and control block. The SIN flag is set to 1 when new data are loaded into DATAIN from the shift register, and cleared to 0 when these data are read by the processor.
- The SOUT flag indicates whether the DATAOUT register is available. It is cleared to 0 when the processor writes new data into DATAOUT and set to 1 when data are transferred from DATAOUT to the output shift register.
- The double buffering used in the input and output a path in Figure is important. It is possible to implement DATAIN and DATAOUT themselves as shift registers, thus obviating the need for separate shift registers.
- After receiving one character from the serial line, the interface would not be able to start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to give the processor time to read the input data.
- With double buffering, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of input data over the serial line. An analogous situation occurs in the output path of the interface.



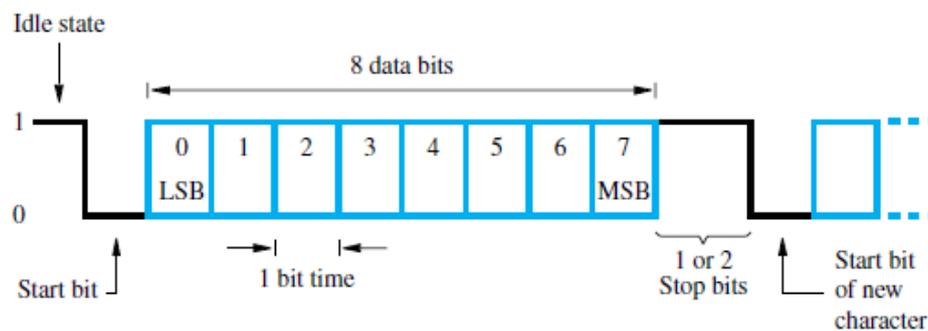
A serial interface

- During serial transmission, the receiver needs to know when to shift each bit into its input shift register. Since there is no separate line to carry a clock signal from the transmitter to the receiver, the timing information needed must be embedded into the transmitted data using an encoding scheme.

- There are two basic approaches. The first is known as asynchronous transmission, because the receiver uses a clock that is not synchronized with the transmitter clock. In the second approach, the receiver is able to generate a clock that is synchronized with the transmitter clock. Hence it is called synchronous transmission.

Asynchronous Transmission

- This approach uses a technique called start-stop transmission. Data are organized in small groups of 6 to 8 bits, with a well-defined beginning and end. In a typical arrangement, alphanumeric characters encoded in 8 bits are transmitted as shown in Figure.
- The line connecting the transmitter and the receiver is in the 1 state when idle. A character is transmitted as a 0 bit, referred to as the Start bit, followed by 8 data bits and 1 or 2 Stop bits. The Stop bits have a logic value of 1.
- The 1-to-0 transition at the beginning of the Start bit alerts the receiver that data transmission is about to begin. Using its own clock, the receiver determines the position of the next 8 bits, which it loads into its input register.
- The Stop bits following the transmitted character, which are equal to 1, ensure that the Start bit of the next character will be recognized. When transmission stops, the line remains in the 1 state until another character is transmitted.



Asynchronous serial character transmission

Synchronous Transmission

- In synchronous transmission, the receiver generates a clock that is synchronized to that of the transmitter by observing successive 1-to-0 and 0-to-1 transitions in the received signal. It adjusts the position of the active edge of the clock to be in the center of the bit position.
- A variety of encoding schemes are used to ensure that enough signal transitions occur to enable the receiver to generate a synchronized clock and to maintain synchronization. Once synchronization is achieved, data transmission can continue indefinitely.
- Encoded data are usually transmitted in large blocks consisting of several hundreds or several thousands of bits. The beginning and end of each block are marked by appropriate codes, and data within a block are organized according to an agreed upon set of rules. Synchronous transmission enables very high data transfer rates.

UNIVERSAL SERIAL BUS (USB):

- The Universal Serial Bus (USB) is the most widely used interconnection standard. A large variety of devices are available with a USB connector, including mice, memory keys, disk drives, printers, cameras, and many more.
- The commercial success of the USB is due to its simplicity and low cost. The original USB specification supports two speeds of operation, called low-speed (1.5 Megabits/s) and full-speed

(12 Megabits/s). Later, USB 2, called High-Speed USB, was introduced. It enables data transfers at speeds up to 480 Megabits/s.

- As I/O devices continued to evolve with even higher speed requirements, USB 3 (called Super speed) was developed. It supports data transfer rates up to 5 Gigabits/s.
- The USB has been designed to meet several key objectives:
 - Provide a simple, low-cost, and easy to use interconnection system
 - Accommodate a wide range of I/O devices and bit rates, including Internet connections, and audio and video applications
 - Enhance user convenience through a “plug-and-play” mode of operation

Device Characteristics

- The kinds of devices that may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from these devices vary significantly.
- In the case of a keyboard, one byte of data is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in a computer system, the data generated by the keyboard are called asynchronous.
- Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 10 bytes per second, which is less than 100 bits per second.
- Consider now a different source of data. Many computers have a microphone, either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically.
- For each sample, an analog-to-digital (A/D) converter generates an n -bit number representing the magnitude of the sample. The number of bits, n , is selected based on the desired precision with which to represent each sample.
- Later, when these data are sent to a speaker, a digital to- analog (D/A) converter is used to restore the original analog signal from the digital format. A similar approach is used with video information from a camera.
- The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time. A signal must be sampled quickly enough to track its highest-frequency components.
- An important requirement in dealing with sampled voice or music is to maintain precise timing in the sampling and replay processes. A high degree of jitter (variability in sample timing) is unacceptable. Hence, the data transfer mechanism between a computer and a music system must maintain consistent delays from one sample to the next. Otherwise, complex buffering and retiming circuitry would be needed.
- Data transfers for images and video have similar requirements, but require much higher data transfer rates.
- To maintain the picture quality of commercial television, an image should be represented by about 160 kilobytes and transmitted 30 times per second. Together with control information, this yields a total bit rate of 44 Megabits/s. Higher-quality images, as in HDTV (High Definition TV), require higher rates.
- Large storage devices such as magnetic and optical disks present different requirements. Their connection to the computer requires a data transfer bandwidth of at least 40 or 50 Megabits/s.

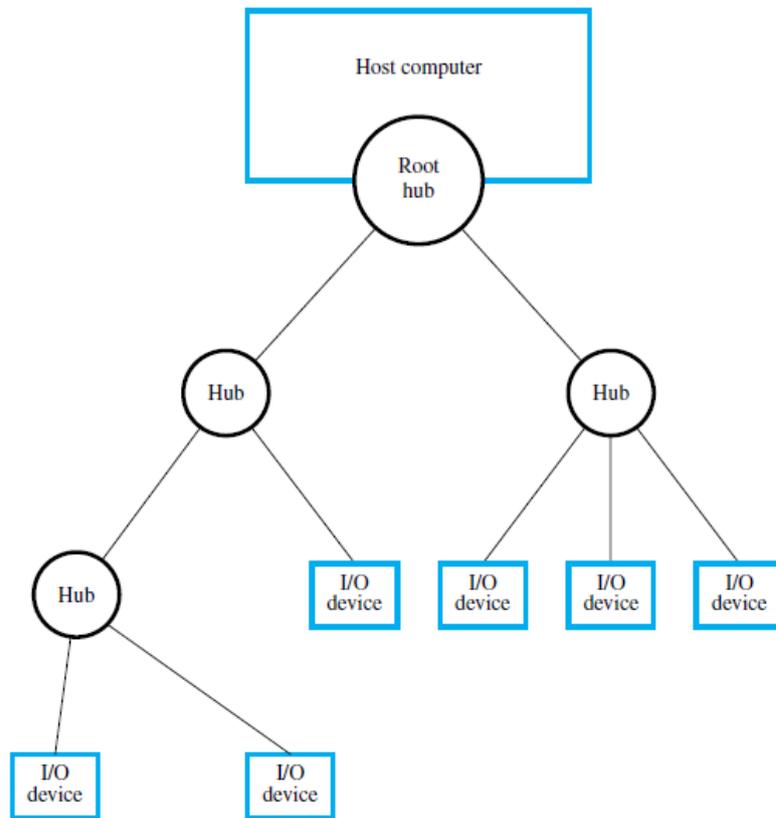
- Delays on the order of milliseconds are introduced by the movement of the mechanical components in the disk mechanism. Hence, a small additional delay introduced while transferring data to or from the computer is not important, and jitter is not an issue. However, the transfer mechanism must guarantee data correctness.

Plug-and-Play

- When an I/O device is connected to a computer, the operating system needs some information about it. It needs to know what type of device it is so that it can use the appropriate device driver. It also needs to know the addresses of the registers in the device's interface to be able to communicate with it.
- The USB standard defines both the USB hardware and the software that communicates with it. Its plug-and-play feature means that when a new device is connected, the system detects its existence automatically.
- The software determines the kind of device and how to communicate with it, as well as any special requirements it might have. As a result, the user simply plugs in a USB device and begins to use it, without having to get involved in any of these details.
- The USB is also hot-pluggable, which means a device can be plugged into or removed from a USB port while power is turned on.

USB Architecture

- The USB uses point-to-point connections and a serial transmission format. When multiple devices are connected, they are arranged in a tree structure as shown in Figure. Each node of the tree has a device called a hub, which acts as an intermediate transfer point between the host computer and the I/O devices.
- At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices: a mouse, a keyboard, a printer, an Internet connection, a camera, or a speaker. The tree structure makes it possible to connect many devices using simple point-to-point serial links.
- If I/O devices are allowed to send messages at any time, two messages may reach the hub at the same time and interfere with each other. For this reason, the USB operates strictly on the basis of polling.
- A device may send a message only in response to a poll message from the host processor. Hence, no two devices can send messages at the same time. This restriction allows hubs to be simple, low-cost devices.
- Each device on the USB, whether it is a hub or an I/O device, is assigned a 7-bit address. This address is local to the USB tree and is not related in any way to the processor's address space. The root hub of the USB, which is attached to the processor, appears as a single device.
- The host software communicates with individual devices by sending information to the root hub, which it forwards to the appropriate device in the USB tree.
- When a device is first connected to a hub, or when it is powered on, it has the address 0. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.
- When the host is informed that a new device has been connected, it reads the information in a special memory in the device's USB interface to learn about the device's capabilities. It then assigns the device a unique
- USB address and writes that address in one of the device's interface registers. It is this initial connection procedure that gives the USB its plug-and-play capability.



Universal Serial Bus tree structure

Isochronous Traffic on USB

- An important feature of the USB is its ability to support the transfer of isochronous data in a simple manner. As mentioned earlier, isochronous data need to be transferred at precisely timed regular intervals.
- To accommodate this type of traffic, the root hub transmits a uniquely recognizable sequence of bits over the USB tree every millisecond. This sequence of bits, called a Start of Frame character, acts as a marker indicating the beginning of isochronous data, which are transmitted after this character. Thus, digitized audio and video signals can be transferred in a regular and precisely timed manner.

Two Marks Questions & Answers

UNIT 1 – BASIC STRUCTURE OF A COMPUTER SYSTEM

1. Write the equation for the dynamic power required per transistor?

$$\text{Power} \propto \frac{1}{2} * \text{capacitor load} * \text{voltage}^2 * \text{Frequency switched}$$

2. Classify the instructions based on the observations they perform and give one example to each category

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20]=\$s1;\$s1=0 or 1	Store word as 2nd half of atomic swap
load upper immed.	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	Loads constant in upper 16 bits	
Logical	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2 20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

3. What are the components of a computer system?

Input, Output, Memory, Datapath and control

4. How to represent Instruction in a computer system?

Instructions are represented in a computer using three formats

R Type for Register (or) R Format

I Type for Immediate (or) I Format

J Type for Jump (or) J Format

5. Distinguish between auto increment and auto decrement addressing mode.

Auto Increment mode	Auto Decrement mode
In this mode the effective address of the operand is the content of a register specified in the instruction. After accessing the operand the content of this register are automatically incremented to point to the next item in a list	In this mode the content of a register specified in the instruction are first automatically decremented and then used effectively.
It can be written as $(Ri)+$	It can be written as $-(Ri)$

6. What are the addressing modes?

- The different ways in which the location of an operand is specified in an instruction is known as addressing modes
- It is a rule for interpreting or translating address field of an instruction into an effective address from where the operand is actually referenced

The addressing modes are the following:

1. Immediate addressing, where the operand is a constant within the instruction itself
2. Register addressing, where the operand is a register
3. Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. PC-relative addressing, where the branch address is the sum of the PC and a constant in the instruction
5. Pseudo direct addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

7. State the Need for indirect addressing mode. Give an example.

In mode, where the operand is at the memory whose address is the sum of register and constant in the instruction. Ex: `lw sto, 32 (ss3)`

8. What is an instruction register (IR)?

An IR is the part of a CPU's control unit that holds the instruction currently being executed or decoded

9. What is instruction set architecture (ISA)?

- ISA is a well-defined hardware/software interface. The contract b/w software & hardware ISA is the part of the computer architecture related to programming including the data types, registers and addressing modes.

10. Give the formula for CPU execution time for a program?

CPU execution time for program=CPU clock cycles for a program x clock cycle time.

11. List out the methods used to improve system performance.

The methods used to improve system performance are 1. Processor clock 2. Basic Performance Equation 3. Pipelining 4.Clock rate 5.Instruction set 6.Compiler.

12. Define Opcode and Operand .

Opcode is the portion of a machine language instruction that specifies the operation to be performed.

Ex: MOV AX, 1000H ; here MOV specifies the movement of 1000H in the AX register.

Operand is a quality on which operation is performed. Here AX and 1000H are the operands.

13. Define MIPS Rate and Throughput Rate.

MIPS Rate: The rate at which the instructions are executed at a given time.

Throughput: The total amount of work done in a given time.

Throughput rate: The rate at which the total amount of work done at a given time.

14. State Amdahl's law.

Amdahl's law, also known as Amdahl's argument, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

15. Define CPI

The term Clock Cycles Per Instruction Which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI. $CPI = \text{CPU clock cycles} / \text{Instruction count}$.

16. State and explain the performance equation?

The average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock cycle rate is R cycles per second, the program execution time is given by $T = (N \times S) / R$ This is often referred to as the basic performance equation.

17. How CPU execution time for a program is calculated?

CPU execution time for a program= CPU clock cycles for a program*Clock Cycle Time

$$\text{CPU execution time for a program} = \frac{\text{CPU Clock cycles for a program}}{\text{Clock Rate}}$$

7. State the representation of double precision floating point number.

- First, if a number is not normalized, it can be put in normalized form by shifting the binary point and adjusting the exponent. Figure shows an unnormalized value, $0.0010110 \dots \times 2^9$, and its normalized version, $1.0110 \dots \times 2^6$. Since the scale factor is in the form 2^i , shifting the mantissa right or left by one bit position is compensated by an increase or a decrease of 1 in the exponent, respectively.
- Second, as computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than -126 or greater than $+127$.
- In the first case, we say that *underflow* has occurred, and in the second case, we say that *overflow* has occurred.



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110 \dots \times 2^9$$

(a) Unnormalized value



$$\text{Value represented} = +1.0110 \dots \times 2^6$$

(b) Normalized version

Floating-point normalization in IEEE single-precision format

8. What is Guard bit? What are the ways to truncate the guard bits?

Guard bits are extra bits which are processed during the intermediate steps to yield maximum accuracy in the final results.

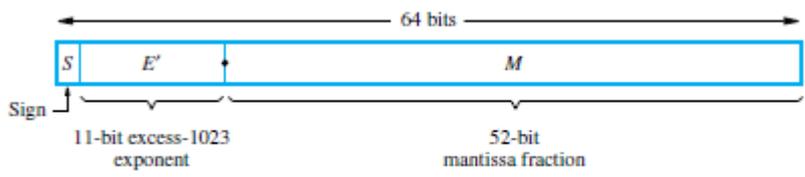
Ways to Truncate:

- Rounding to positive infinity
- Rounding to negative infinity
- Rounding to zero
- Rounding to Even Number

9. In floating point numbers when do you say that an underflow or overflow has occurred.

In single precision numbers when an exponent is less than -126 then we say that an underflow has occurred. When an exponent is less than $+126$ then we say that an over flow has occurred.

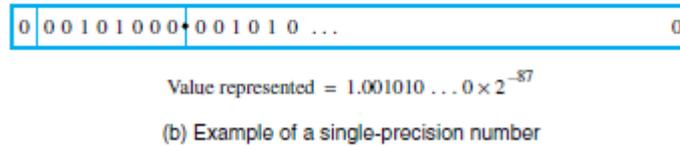
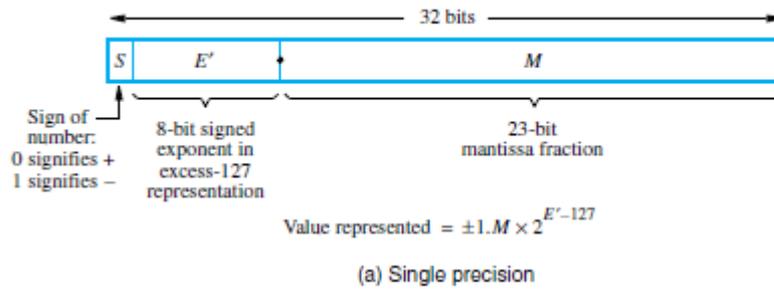
10. Give the representation of double precision floating point number



$$\text{Value represented} = \pm 1.M \times 2^{E-1023}$$

(c) Double precision

11. Give the IEEE standard for floating point number for single precision model.



12. What is excess-127 format?

Instead of the signed exponent E, the value actually stored in the exponent field is an unassigned integer $E^I = E + 127$. This format is called 127 format.

13. What is a carry look-ahead adder?

The input carry needed by a stage is directly computed from carry signals obtained from all the preceding stages $i-1, i-2, \dots, 0$, rather than waiting for normal carries to supply slowly from stage to stage. An adder that uses this principle is called a carry look-ahead adder.

14. What are the main features of Booth's algorithm?

- 1) It handles both positive and negative multipliers uniformly.
- 2) It achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s

15. What are the 2 ways to detect overflow in an n-bit adder?

Overflow can occur when the signs of two operands are the same. Overflow occurs when the carry bits C_n and C_{n-1} are different.

UNIT 3 PROCESSOR & CONTROL UNIT

1. Define a data path in a CPU.

A unit used to operate on or hold data within a processor. In the MIPS implementation, the data path elements include the instruction and data memories, the register file, the ALU, and adders.

2. Name the control signals required to perform arithmetic operations.

- (a) Reg Not (b) Reg write (c) ALL src (d) PCsrc (e) Mem Read (f) Mem to Reg

3. Define hazard. Give an example for data hazard.

Any condition that causes the pipeline to stall is called a hazard. Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

4. What is meant by pipeline bubble?

Pipeline bubble or pipeline stall is a delay in execution of an instruction in an instruction pipeline in order to resolve the hazard. During the decoding stage, the control unit will determine if the decoded instruction reads from a register that the instruction currently in the

5. What is the ideal CPI of a pipelined processor?

The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} = 1 + \text{Pipeline stall clock cycles per instruction}$$

6. Mention the various types of pipelining.

It is divided into 2 categories:

- Arithmetic Pipeline
- Instruction Pipeline

7. Mention the various phase in executing an instruction.

- Fetch
- Decode
- Execute
- Memory
- Write Back

8. What are the advantages of pipelining?

The cycle time of the processor is reduced, thus increasing instruction issue rate in most cases. Some combinational circuits such as adders or multipliers can be made faster by adding more circuitry. If pipelining is used instead it can save circuitry and also a more complex combinational circuit.

9. What is meant by branch prediction?

Branch prediction, Predict the next fetch address. There are two branch prediction techniques:

- Static branch prediction
- Dynamic branch prediction

The performance of branch prediction technique depends on

- Accuracy
- Cost

10. Give the features of the addressing modes suitable for pipelining.

The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory
- Only load and store instructions access memory operands
- The addressing modes used do not have side effects

11. What is loop unrolling?

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is loop unrolling. Unrolling simply replicates the loop body multiples times, adjusting the loop termination code.

12. What is the role of cache memory in pipeline?

The use of cache memory is to solve the memory access problem. When cache is included in the processor the access time to the cache is usually the same time needed to perform other basic operation inside the processor.

13. Name the methods for generating the control signals.

The methods for generating the control signals are:

- 1) Hardwired control
- 2) Microprogrammed control

14. What are the two main approaches to hardware multithreading?

There are two main approaches to hardware multithreading. Fine-grained multithreading switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. Coarse-grained multithreading is an alternative to fine-grained multithreading. It switches threads only on costly stalls, such as last-level cache misses.

UNIT 4 – PARALLELISIM

1. What is instruction level parallelism?

Pipelining exploits the potential parallelism among instructions. This Parallelism is called instruction-level parallelism (ILP).

2. State the need for ILP.

It provides parallel scheduling which is divided into two types

- (a) Hardware solutions
- (b) Software solutions

3. What is multithreading?

Multithreading allows multiple threads to share the functional units of the single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread.

4. Distinguish implicit and explicit multithreading.

Implicit multithreading refers to the ability of the compiler to perform certain operation in parallel without any hint from the programmer.

Explicit multithreading is achieved by programmer that can

5. What is fine grained multithreading?

Fine grained multithreading switches between threads on each instruction causing the execution of multiple threads to be interleaved. This interleaving is often done in round robin fashions, skipping any threads that are stalled at that time.

6. Define strong scaling and weak scaling.

- **Strong scaling** means measuring speed-up while keeping the problem size fixed.
- **Weak scaling** means that the problem size grows proportionally to the increase in the number of processors.

7. Difference between fine grained multithreading and coarse-grained multithreading.

Fine grained multithreading	Coarse-grained multithreading
Switches between threads on each instruction, resulting in interleaved execution of multiple threads	Switches threads only on costly stalls, such as last-level cache misses
It can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls	It is limited in its ability to overcome throughput losses, especially from shorter stalls.
It slows down the execution of the individual threads	It is much less likely to slow down the execution of an individual thread

8. What is meant by exception? Give one example of MIPS exception.

Exception is an unscheduled event that disrupts program execution; used to detect overflow.

Or

An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

add \$S₁, \$S₂, \$S₁,

9. What is meant by address mapping?

Address mapping is defined as the smallest unit of addressed data that can be mapped independently in an area of the virtual address space.

10. Protein string matching code has four days execution time on current machine doing integer instructions in 20 % of time, doing I/O in 35% of time and other operations in the remaining time. Which is the better trade off among the following two proposals? First: Compiler optimization that reduces number of integer instructions by 25% (assume each integer instruction takes the same amount of time); Second: Hardware optimization that reduces the latency of each I/O operations from 6µs to 5 µs.

Solution:

4 days execution time on current machine

20% of time doing integer instructions

35% of time doing I/O

Speed up integer ops

X=0.2

S= (1/1-0.25)=1.33

S_{int} =1/(0.2/1.133+0.8) = 1.052

Speed up IO

X=0.35

S=6µs/5 µs = 1.2

Speeding Up IO is better

11. Give example for each class in Flynn's classification.

SISD: Traditional Uniprocessor

SIMD: The intel Pentium 3

MISD: Multiple Frequency filters operating on a single signal stream

MIMD: Sun Ultra Services

12. What is Sub word Parallelism?

Sub word Parallelism, we pack multiple sub words into a word and then process whole words. With the appropriate sub word boundaries, this technique results in parallel processing of sub words. Since the same instruction applies to all the sub words within the word, this is a form of small scale SIMD processing.

13. Define a super scalar processor.

Super scalar processors attempts to issue multiple instruction per cycle. However, essential dependencies are specified by sequential ordering so operations must be processed in sequential order.

14. Give the key characteristics of GPUs from CPUs:

- GPUs are accelerators that supplement a CPU, so they do not need be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.
- The GPU problems sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

15. What is Cluster?

Clusters are generally collections of computer connected to each other over their I/O interconnect via standard network switches and cables. Clusters are the best example of message passing parallel computer.

16. What are the three major distinctions Warehouse Scale computers have?

- Ample, easy parallelism
- Operational Costs Count
- Scale and the Opportunities/Problems Associated with Scale

17. Compare UMA and NUMA Multiprocessors.

Uniform Memory Access(UMA) Multiprocessors: A Multiprocessors in which latency to any word in main memory is about the same no matter which processor requests the access.

Non- Uniform Memory Access(NUMA) Multiprocessors: A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

18. What are the classifications made by Flynn's?

The classifications defined by Flynn are based upon the number of concurrent instruction (or control) and data streams available in the architecture.

1. Single Instruction, Single Data stream (SISD)
2. Single Instruction ,Multiple data stream(SIMD)
3. Multiple Instruction, Single Data stream (MISD)
4. Multiple Instruction, Multiple Data streams (MIMD)
5. single program, multiple data (SPMD)

19. what is the basic philosophy of Vector Architecture?

The basic philosophy of vector architecture is to collect data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory.

20. Define task level parallelism.

High performance can mean high throughput for independent task. Utilizing multiple processors by running independent programs simultaneously is called as **task level parallelism** or process-level parallelism.

UNIT 5 – MEMORY & I/O SYSTEMS

1. What are the various memory technologies?

- Main memory-SRAM semiconductor memory
- Main memory-DRAM semiconductor memory
- Flash semiconductor memory
- Magnetic disk

2. Define Hit Ratio.

Hit Ratio is the fraction memory access found in the upper level .It is often used as a measure of the performance of the memory hierarchy.

3. Distinguish SRAM and DRAM.

SRAMs are simply integrated circuits that are memory arrays with the single access port that can provide either read or a write. SRAMs have a fixed access time to any datum. SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use 6 to 8 transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In a DRAM the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to over write the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAM store charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed.

4. What is virtual memory?

A technique that uses main memory as a “cache” for secondary storage. Two major motive for Virtual Memory

- To allow efficiency and safe sharing of memory among multiple programs
- To remove the programming burdens of a small, limited amount of main memory

5. Define memory hierarchy.

In computer architecture memory hierarchy is a concept used for storing and discussing performance issues in computer architectural design, algorithm predictions and the lower level programming constructs such as involving locality of reference. The memory hierarchy in computer storage distinguishes each level in the hierarchy by response time. Since response time, complexity and capacity are related, the levels may also be distinguished by their performance and controlling technologies.

6. State the advantages of virtual memory

- Easier memory management
- Provides memory isolation/ protection

7. What is cache memory?

Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAMs

8. Define memory interleaving.

Memory Interleaving is a design made to compensate for the relatively slow speed of dynamic RAM by spreading memory address evenly across memory banks.

9. Summarize the sequence of the events involved in handling an interrupt request from a single device.

- The device raises an interrupt request
- The processor interrupts the program currently being executed
- Interrupts are disabled by changing the control bits in the PS
- The action requested by the interrupts is performed by ISR
- Interrupts are enabled and execution of the interrupted program is resumed

10. How many total bits are required for a direct map cache with 16KB of data and 4-word blocks, assuming a 32bit address?

Solution:

We know that 16KB is 4096, 4K words is 2^{12} words.

Block size of 4 words (2^2), there are 1024 (2^{10}) blocks.

Each block has $4 \times 32 = 128$ bits of data plus a tag.

Thus the total cache size is: $210 \times (128 + (32 - 10 - 2 - 2) + 1) = 210 \times 147 = 147$ bits

11. What is the use of DMA controller?

- Used for high speed I/O devices
- Device interface transfers data directly to or from the memory
- Processor not continuously involved

12. What is miss rate?

The **miss rate (1-hit rate)** is the fraction of memory accesses not found in the upper level.

13. State the advantages of the Virtual Memory.

1. Virtual memory makes application programming easier by hiding fragmentation of physical memory.

2. We can run more applications at once.

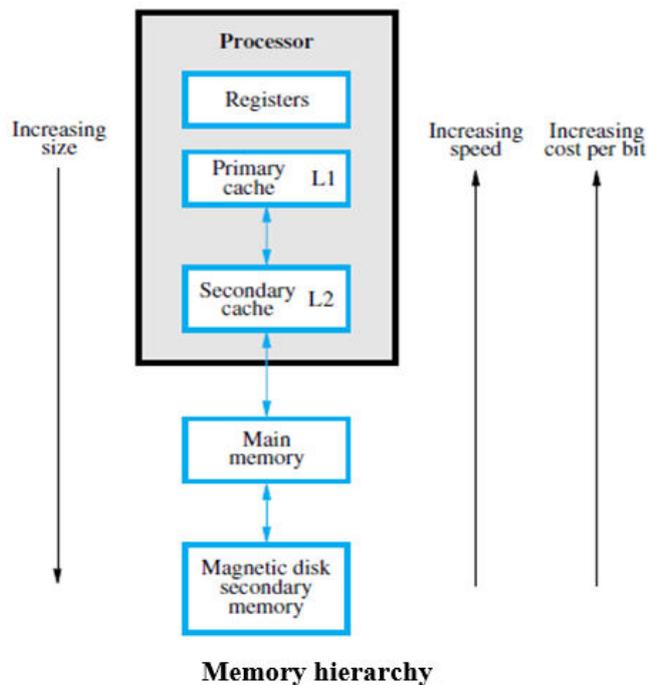
14. Differentiate Programmed I/O and Interrupt I/O

Sl. No	Programmed I/O	Interrupt I/O
1	During polling processor is busy and therefore have serious and decremental effect on system throughput.	Here the processor is allowed to execute its instruction in sequence and only stop to service I/O device when it is told to do so by the device itself. This increase system throughput.
2	It is implemented without interrupt hardware support	It is implemented using interrupt hardware support.
3	It does not depend on interrupt status.	Interrupt must be enabled to process
4	It does not need initialization of stack	It needs initialization of stack
5	System throughput decreases as number of I/O devices connected in the system increases	System throughput does not depend on number of I/O devices connected in the system.

15. What do you mean by memory mapped I/O?

In Memory mapped I/O, there is no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words.

16. Draw Memory Hierarchy in a typical computer system.



7. Web server is to be enhanced with a new CPU which is 10 times faster on computation than old CPU. The original CPU spent 40% of its time processing and 60% of its time waiting for I/O. What will be the overall speedup?
8. Classify shared memory multiprocessor based on the memory access latency.
9. Draw the memory hierarchy in a typical computer system.
10. What is meant by memory-mapped I/O?

PART B — (5 × 13 = 65 marks)

11. (a) (i) Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (class A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3 respectively, and P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2 respectively. Given a program with a dynamic instruction count of 1.0×10^6 instructions divided into classes as follows: 10% class A, 20% class B, 50% class C, and 20% class D, which implementation is faster? What is the global CPI for each implementation? Find the clock cycles required in both cases. (7)
- (ii) Explain the three broad classes of applications of computers. (6)

Or

- (b) (i) Assume that the variables f and g are assigned to registers $\$s0$ and $\$s1$ respectively. Assume that the base address of the array A is in register $\$s2$. Assume f is zero initially.

$$f = -g - A[4]$$

$$A[5] = f + 100;$$
 Translate the above C statements into MIPS code. How many MIPS assembly instructions are needed to perform the C statements and how many different registers are needed to carry out the C statements? (5)
 - (ii) Define addressing mode in a computer. What are the different MIPS addressing modes? Give one example instruction to each category. (8)
12. (a) (i) Multiply the following signed numbers using Booth algorithm. $A = (-34)_{10} = (1011110)_2$ and $B = (22)_{10} = (0010110)_2$ where B is multiplicand and A is multiplier. (6)
 - (ii) Draw the block diagram of integer divider and explain the division algorithm. (7)

Or

- (b) (i) How IEEE 752 32-bit single precision floating point numbers are represented? Example. How are print numbers represented? (3)
- (ii) Explain floating point addition algorithm with a neat block diagram? (10)
13. (a) Draw a simple MIPS datapath with the control unit and explain the execution of ALU instructions. (13)

Or

- (b) (i) A processor has five individual stages, namely, IF, ID, EX, MEM, and WB and their latencies are 250ps, 350ps, 150ps, 300ps, and 200ps respectively. The frequency of the instructions executed by the processor are as follows ; ALU : 40%, Branch : 25%, load : 20% and store:15% What is the clock cycle time in a pipelined and non-pipelined processor? If you can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor? Assuming there are no stalls or hazards, what is the utilization of the data memory? Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit? (6)
- (ii) List the hazards in pipelining a processor and give one example for each. (7)
14. (a) (i) List the software and hardware techniques to achieve Instruction Level Parallelism (ILP). (4)
- (ii) Discuss the challenges in parallel processing in enhancing computer architecture. (9)

Or

- (b) (i) Explain any three types of hardware multithreading. (9)
- (ii) Define the classes in Flynn's Taxonomy of computer architectures. Give one example for each class. (4)
15. (a) (i) Discuss the three mapping techniques in memory hierarchy. Explain with examples. (10)
- (ii) Define Translation Lookaside Buffer (TLB). What is its use? (3)

Or

- (b) Explain mechanisms Direct Memory Access and Interrupt handling. (6 + 7)

3

20360

PART C — (1 × 15 = 15 marks)

16. (a) (i) Consider web browsing application. Assuming both client and server are involved in the process of web browsing application, where can caches be placed to speed up the process? Design a memory hierarchy for the system. Show the typical size and latency at various levels of the hierarchy. What is the relationship between cache size and its access latency? What are the units of data transfers between hierarchies? What is the relationship between the data location, data size, and transfer latency?
- (ii) The following sequence of instructions are executed in the basic 5-stage pipelined processor:
- lw \$1, 40(\$6)
- add \$6, \$2, \$2
- sw \$6, 50(\$1)
- Indicate dependences and their type. Assuming there is no forwarding in this pipelined processor, indicate hazards and add NOP instructions to eliminate them.

Or

- (b) Compare hardwired and microprogrammed control unit designs in terms of their mechanism of generating control signals with diagram. (15)



Reg. No. :

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Question Paper Code : 40902

R. E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2018

Third/Fifth/Sixth Semester

Computer Science and Engineering

CS 6303 – COMPUTER ARCHITECTURE

(Common to : Electronics and Communication Engineering/Electronics and Instrumentation Engineering/Instrumentation and Control Engineering/Robotics and Automation Engineering/Information Technology)
(Regulations 2013)

Time : Three Hours

Maximum : 100 Marks

Answer ALL questions

PART – A

(10×2=20 Marks)

1. Write the equation for the dynamic power required per transistor.
2. Classify the instructions based on the operations they perform and give one example to each category.
3. Show the IEEE 754 binary representation of the number $(-0.75)_{10}$ in single precision.
4. Define a datapath in a CPU.
5. What is the ideal CPI of a pipelined processor ?
6. What is meant by exception ? Give one example of MIPS exception.
7. Protein String Matching Code has 4 days execution time on current machine doing integer instructions in 20% of time, doing I/O in 35% of time and other operations in the remaining time. Which is the better tradeoff among the following two proposals ? First : Compiler optimization that reduces number of integer instructions by 25% (assume each integer instruction takes the same amount of time); Second : Hardware optimization that reduces the latency of each IO operations from $6\mu s$ to $5\mu s$.
8. Give example for each class in Flynn's classification.
9. Distinguish SRAM and DRAM.
10. What is the use of DMA Controller ?



PART - B

(5×13=65 Marks)

11. a) i) Consider three different processors P1, P2 and P3 executing the same instruction set. P1 has a 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2.
- a) Which processor has the highest performance expressed in instructions per second?
- b) If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions in each processor. (5)
- ii) Explain in detail the components of a computer system. (8)

(OR)

- b) i) Translate the following C code to MIPS assembly code. Use a minimum number of instructions. Assume that i and k correspond to registers \$s3 and \$s5 and the base of the array save is in \$s6. What is the MIPS assembly code corresponding to this C segment?
- ```
while (save[i] == k)
 i += 1;
```
- (5)
- ii) What is an addressing mode in a computer? Classify MIPS addressing modes and give one example instruction to each category. (8)
12. a) i) Perform  $X + Y$  and  $Y - X$  using 2's complements for given the two binary numbers  $X = 0000\ 1011\ 1110\ 1111$  and  $Y = 1111\ 0010\ 1001\ 1101$ . (5)
- ii) Multiply the following signed 2's complement numbers using the Booth algorithm.  $A = 001110$  and  $B = 111001$  where A is multiplicand and B is multiplier. (8)

(OR)

- b) i) Draw the block diagram of integer divider and explain the division algorithm. (5)
- ii) Add the numbers  $(0.75)_{10}$  and  $(-0.275)_{10}$  in binary using the Floating point addition algorithm. (8)



13. a) Design a simple datapath with the control unit and explain in detail. (13)

(OR)

b) Discuss the limitations of pipelining a processor's datapath. Suggest the methods to overcome them. (13)

14. a) i) List the limitations of instruction level parallelism. (5)  
ii) What are the challenges in parallel processing? (8)

(OR)

b) i) Compare and contrast fine-grained multi-threading, coarse-grained multi-threading and simultaneous multi-threading. (9)  
ii) Classify shared memory multiprocessor based on the memory access latency. (4)

15. a) i) What is the need for Cache memory? List the three mapping methods of Cache memory and explain any two. (10)  
ii) Define virtual memory. What is the advantage of using virtual memory? (3)

(OR)

b) i) Discuss about Programmed I/Os associated with computers. (6)  
ii) Write the sequence of operations carried out by a processor when interrupted by a peripheral device connected to it. (7)

PART - C

(1×15=15 Marks)

16. a) The following sequence of instructions are executed in the basic 5-stage pipelined processor :

or r1, r2, r3

or r2, r1, r4

or r1, r1, r2

a) Indicate dependences and their type.

b) Assume there is no forwarding in this pipelined processor. Indicate hazards and add NOP instructions to eliminate them.

c) Assume there is full forwarding. Indicate hazards and add NOP instructions to eliminate them. (15)

(OR)

b) Explain the detail of DMA control with suitable diagrams. Discuss how it improve the overall performance of the system. (15)



50384



12. a) Illustrate the division algorithm with an example. (13)  
(OR)
- b) i) Add the numbers  $(0.5)_{10}$  and  $(0.4375)_{10}$  using the floating point addition. (6%)  
ii) Multiply the numbers  $(0.5)_{10}$  and  $(0.4375)_{10}$  using the floating point multiplication. (6%)
13. a) Explain in detail the operation of the data path. (13)  
(OR)
- b) Explain the pipeline hazard in detail. (13)
14. a) Explain with diagrammatic illustration Flynn's classification. (13)  
(OR)
- b) Describe Simultaneous Multithreading (SMT) with an example. (13)
15. a) Explain in detail about the memory technologies. (13)  
(OR)
- b) What is cache memory? How to improve cache performance? Discuss. (13)

PART - C

(1×15=15 Marks)

16. a) i) Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of the original computation can be sequential? (8)  
ii) Suppose you want to perform two sums: one is a sum of 10 scalar variables and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20. (7)  
(OR)
- b) Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%. (15)

**Question Paper Code : 71675**

B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2017

Third/Fifth/Sixth Semester

Computer Science and Engineering

CS 6303 — COMPUTER ARCHITECTURE

(Common to Electronics and Communication Engineering, Electronics and Instrumentation Engineering, Instrumentation and Control Engineering, Robotics and Automation Engineering, Information Technology)

(Regulations 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)



1. List the major components of a computer system.
2. State the need for indirect addressing mode. Give an example.
3. Subtract  $(11010)_2 - (10000)_2$  using 1's complement and 2's complement method.
4. Write the rules to perform addition on floating point numbers.
5. Name the control signals required to perform arithmetic operations.
6. Define hazard. Give an example for data hazard.
7. What is instruction level parallelism?
8. Distinguish implicit multithreading and explicit multithreading.
9. Define memory interleaving.
10. Summarize the sequence of events involved in handling an interrupt request from a single device.

PART B — (5 × 13 = 65 marks)

11. (a) Explain the important measures of the performance of a computer and derive the basic performance equation. (13)

Or

- (b) Explain direct, immediate, relative and indexed addressing modes with examples. (13)

12. (a) (i) Demonstrate multiplication of two binary numbers with an example. Design an arithmetic element to perform this multiplication. (7)

- (ii) Describe non restoring division with an example. (6)



Or

- (b) (i) Design an arithmetic element to perform the basic floating point operations. (7)

- (ii) What is meant by sub word parallelism? Explain. (6)

13. (a) Discuss the modified data path to accommodate pipelined executions with a diagram. (13)

Or

- (b) (i) Explain the hazards caused by unconditional branching statements. (7)

- (ii) Describe operand forwarding in a pipeline processor with a diagram. (6)

14. (a) (i) Discuss the challenges in parallel processing with necessary examples. (6)

- (ii) Explain Flynn's classification of parallel processing with necessary diagrams. (7)

Or

- (b) Explain the four principal approaches to multithreading with necessary diagrams. (13)

15. (a) Explain the different mapping functions that can be applied on cache memories in detail. (13)

Or

2

71675

- (b) (i) Explain virtual memory address translation in detail with necessary diagrams. (7)

- (ii) What is meant by Direct Memory Access? Explain the use of DMA controllers in a computer system. (6)

PART C — (1 × 15 = 15 marks)

16. (a) (i) Explain mapping functions in cache memory to determine how memory blocks are placed in cache. (8)

- (ii) Explain in detail about the Bus Arbitration techniques in DMA. (7)

Or

- (b) A pipelined processor uses delayed branch technique. Recommend any one of the following possibility for the design of the processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot. In the second possibility, it has a 6-stage pipeline and two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20% of the instructions are branch instructions and that an optimizing compiler has an 80% success rate in filling in the single delay slot. For the second alternative, the compiler is able to fill the second slot 25% of the time.

Reg. No. :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

**Question Paper Code : 80289**

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2016.

Sixth Semester

Electronics and Communication Engineering

CS 6303 — COMPUTER ARCHITECTURE

(Common to Third Semester Information Technology and Computer Science and Engineering)

(Regulations 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is an instruction register?
2. Give the formula for CPU execution time for a program.
3. What is a guard bit and what are the ways to truncate the guard bits?
4. What is arithmetic overflow?
5. What is meant by pipeline bubble?
6. What is a data path?
7. What is instruction level parallelism?
8. What is multithreading?
9. What is meant by address mapping?
10. What is cache memory?

PART B — (5 × 13 = 65 marks)

11. (a) Explain in detail the various components of computer system with neat diagram.

Or

- (b) Explain the different types of Addressing modes with suitable examples.
12. (a) Explain Booth's Algorithm for the multiplication of signed two's complement numbers.

Or

- (b) Discuss in detail about division algorithm in detail with diagram and examples.
13. (a) Why is branch prediction algorithm needed? Differentiate between the static and dynamic techniques.

Or

- (b) Explain how the instruction pipeline works. What are the various situations where an instruction pipeline can stall?
14. (a) Explain in detail about Flynn's classification of parallel hardware.

Or

- (b) Discuss Shared memory multiprocessor with a neat diagram.

15. (a) Discuss DMA controller with block diagram.

Or

- (b) Discuss the steps involved in the address translation of virtual memory with necessary block diagram.

PART C — (1 × 15 = 15 marks)

16. (a) What is the disadvantage of Ripple carry addition and how it is overcome in carry look ahead adder and draw the logic circuit CLA.

Or

- (b) Design and explain a parallel priority interrupt hardware for a system with eight interrupt sources.



**PART – B (5 × 16 = 80 Marks)**

11. (a) Discuss about the various components of a computer system. (16)

**OR**

(b) Elaborate the different types of addressing modes with a suitable example. (16)

12. (a) Explain briefly about floating point addition and Subtraction algorithms. (16)

**OR**

(b) Define Booth Multiplication algorithm with suitable example. (16)

13. (a) What is pipelining ? Discuss about pipelined data path and control. (16)

**OR**

(b) Briefly explain about various categories of hazards with examples. (16)

14. (a) Explain in detail about Flynn's classification. (16)

**OR**

(b) Write short notes on : (16)

(i) Hardware multithreading

(ii) Multicore processors.

15. (a) Define Cache Memory ? Explain the Various Mapping Techniques associated with cache memories. (16)

**OR**

(b) Explain about DMA controller, with the help of a block diagram. (16)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

**Question Paper Code : 27162**

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2015.

Third Semester

Computer Science and Engineering

CS 6303 — COMPUTER ARCHITECTURE

(Common to Information Technology)

(And also common to Fifth Semester Elective – Electronics and Instrumentation Engineering, Instrumentation and Control Engineering and Fifth Semester – Robotics and Automation Engineering )

(Regulations 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What is Instruction set architecture?
2. How CPU execution time for a program is calculated?
3. What are the overflow/underflow conditions for addition and subtraction?
4. State the representation of double precision floating point number.
5. What is a hazard? What are its types?
6. What is meant by branch prediction?
7. What is ILP?
8. Define a super scalar processor.
9. What are the various memory technologies?
10. Define Hit ratio.

PART B — (5 × 16 = 80 marks)

11. (a) Explain in detail the various components of computer system with neat diagram. (16)

Or

- (b) What is an addressing mode? Explain the various addressing modes with suitable examples. (16)

12. (a) Explain in detail about the multiplication algorithm with suitable example and diagram. (16)

Or

- (b) Discuss in detail about division algorithm in detail with diagram and examples. (16)

13. (a) Explain the basic MIPS implementation with necessary multiplexers and control lines. (16)

Or

- (b) Explain how the instruction pipeline works? What are the various situations where an instruction pipeline can stall? Illustrate with an example. (16)

14. (a) Explain in detail Flynn's classification of parallel hardware. (16)

Or

- (b) Explain in detail about hardware Multithreading. (16)

15. (a) What is virtual memory? Explain in detail about how virtual memory is implemented with neat diagram? (16)

Or

- (b) Draw the typical block diagram of a DMA controller and explain how it is used for direct data transfer between memory and peripherals? (16)



12. (a) Explain the sequential version of multiplication algorithm and its hardware. (16)

Or

- (b) Explain how floating point addition is carried out in a computer system. Give an example for a binary floating point addition.

13. (a) Explain the different types of pipeline hazards with suitable examples (16)

Or

- (b) Explain in detail how exceptions are handled in MIPS architecture? (16)

14. (a) Discuss about SISD, MIMD, SIMD, SPMD and VECTOR systems. (16)

Or

- (b) What is hardware multithreading? Compare and contrast Fine grained Multi-Threading and Coarse grained Multi-Threading. (16)

15. (a) Elaborate on the various memory technologies and its relevance. (16)

Or

- (b) What is virtual memory? Explain the steps involved in virtual memory address translation. (16)

**Question Paper Code : 97046**

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2014.

Third Semester

Computer Science and Engineering

CS 6303 — COMPUTER ARCHITECTURE

(Common to Information Technology)

(Regulation 2013)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. State Amdahl's Law.
2. Brief about Relative addressing mode with an example.
3. Define Little Endian arrangement.
4. What is DMA?
5. What is the need for Speculation?
6. What is Exception?
7. What is Flynn's Classification?
8. Brief about Multithreading.
9. Differentiate Programmed I/O and Interrupt I/O.
10. What is the purpose of Dirty/Modified bit in Cache memory?

11. (a) (i) Assume a two address format specified as source, destination. Examine the following sequence of instructions and explain the addressing modes used and the operation done in every instruction. (10)

- (1) Move (R5)+, R0
- (2) Add (R5)+, R0
- (3) Move R0, (R5)
- (4) Move 16(R5), R3
- (5) Add #40, R5.

- (ii) Consider the computer with three instruction classes and CPI measurements as given below and Instruction counts for each instruction class for the same program from two different compilers are given. Assume that the computer's clock rate is 4GHZ. Which Code sequence will execute faster according to execution time? (6)

| Code from  | CPI for this Instruction Class   |   |   |
|------------|----------------------------------|---|---|
|            | A                                | B | C |
| CPI        | 1                                | 2 | 3 |
| Code from  | Instruction Count for each Class |   |   |
|            | A                                | B | C |
| Compiler 1 | 2                                | 1 | 2 |
| Compiler 2 | 4                                | 1 | 1 |

Or

- (b) (i) Explain the components of a computer System. (8)
- (ii) State the CPU performance equation and discuss the factors that affect performance. (8)
12. (a) (i) Multiply the following pair of signed nos. using Booth's bit-pair recoding of the multiplier. A = +13 (Multiplicand) and B = -6 (Multiplier). (10)
- (ii) Briefly Explain Carry lookahead adder. (6)

Or

- (b) Divide  $(12)_{10}$  by  $(3)_{10}$  using the Restoring and Non restoring division algorithm with step by step intermediate results and explain. (16)
13. (a) Explain Data path and its control in detail. (16)
- Or
- (b) What is Hazard? Explain its types with suitable examples. (16)

14. (a) Explain Instruction level Parallel Processing. State the challenges of parallel Processing. (16)

Or

- (b) Explain the terms:
- (i) Multicore Processor. (16)
  - (ii) Hardware Multithreading.
15. (a) (i) Explain mapping functions in cache memory to determine how memory blocks are placed in Cache. (8)
- (ii) Explain in detail about the Bus Arbitration techniques in DMA. (8)

Or

- (b) (i) Draw different memory address layouts and brief about the technique used to increase the average rate of fetching words from the main memory. (8)
- (ii) Explain in detail about any two Standard Input and Output Interfaces required to connect the I/O device to the Bus. (8)