UNIT-I

COMBINATIONAL LOGIC

Combinational circuits-KMap-Analysis and Design Procedures-Binary Adder-Binary Adder-Decimal Adder- Magnitude comparator-Decoder-Encoder-Multiplexers-Demultiplexers

INTRODUCTION:

The digital system consists of two types of circuits, namely

(*i*) Combinational circuits

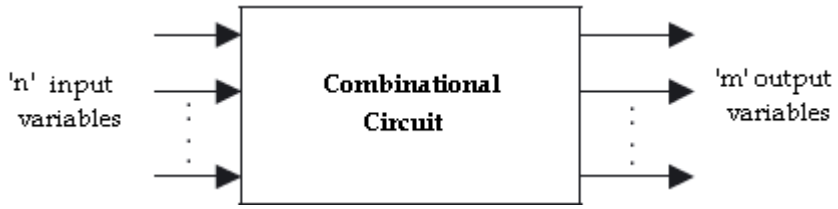
(*ii*) Sequential circuits

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 PROCEDURES:

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.

Problems:

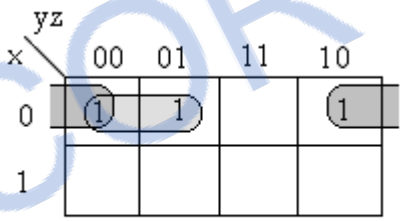
1. Design a combinational circuit with three inputs and one output. The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.

Solution:

Truth Table:

x	y	z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

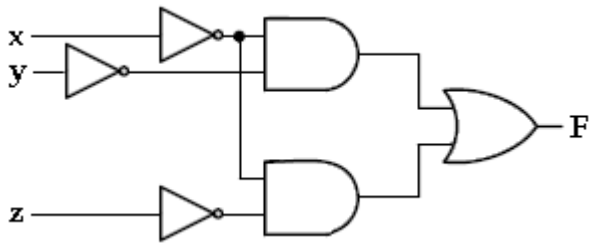
K-map Simplification:



$= x' y' + x' z'$

Logic Diagram:

The combinational circuit can be drawn as,



2. Design a combinational circuit with three inputs, x, y and z, and the three outputs, A, B, and C. when the binary input is 0, 1, 2, or 3, the binary output is

one greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is one less than the input.

Solution:

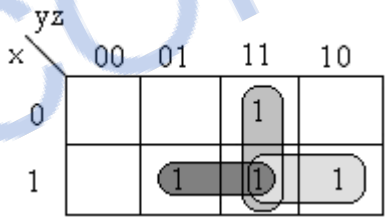
Truth Table:

Derive the truth table that defines the required relationship between inputs and outputs.

x	y	z	A	B	C
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

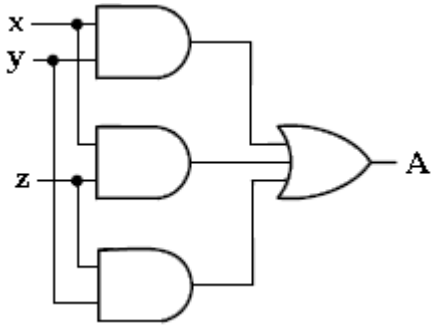
Obtain the simplified Boolean functions for each output as a function of the input variables.

K-map for output A:

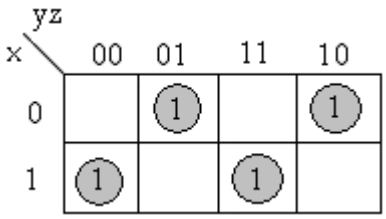


The simplified expression from the map is: **A= xz+ xy+ yz**

Logic Diagram**:**

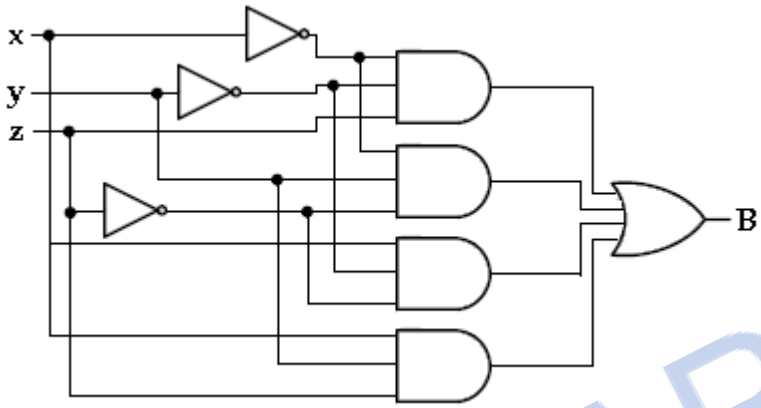


K-map for output B:

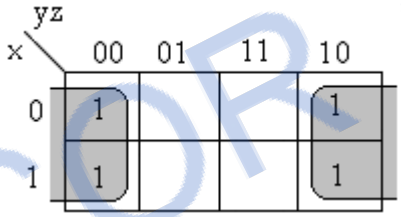


The simplified expression from the map is: **B= x′y′z+ x′yz′+ xy′z′+ xyz**

Logic Diagram**:**

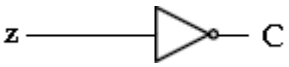


K-map for output C:



The simplified expression from the map is: **C=z′**

Logic Diagram**:**



3. A majority circuit is a combinational circuit whose output is equal to 1 if the input variables have more 1′s than 0′s. The output is 0 otherwise. Design a 3-input majority circuit.

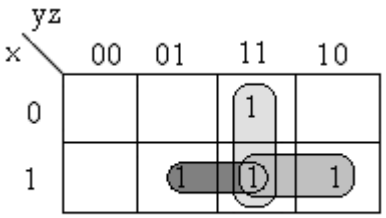
Solution:

Truth Table:

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

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

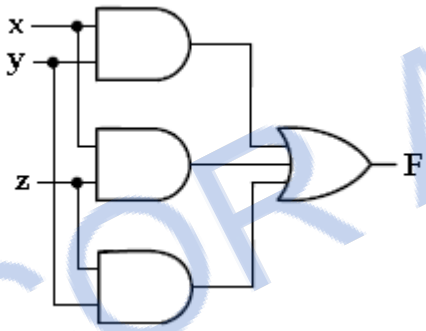
K-map Simplification:



F= xz+ yz+ xy

The simplified expression from the map is: **xz+ yz+ xy**

Logic Diagram:



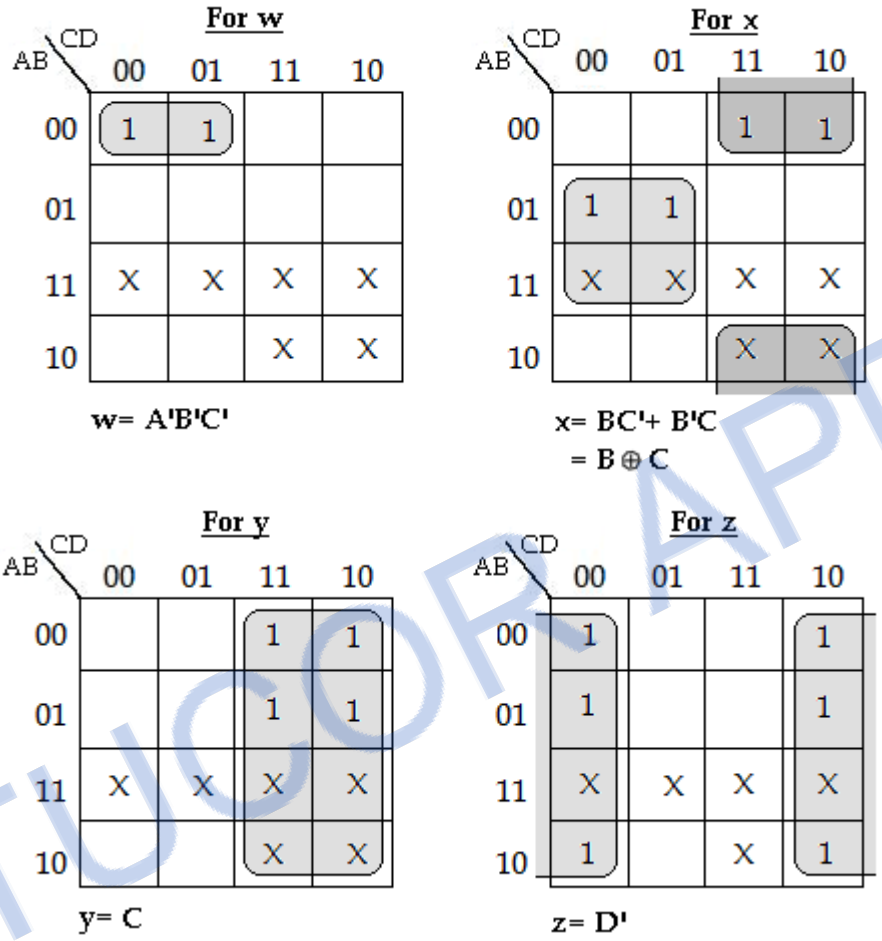
4. Design a combinational circuit that generates the 9's complement of a BCD digit.

Solution:

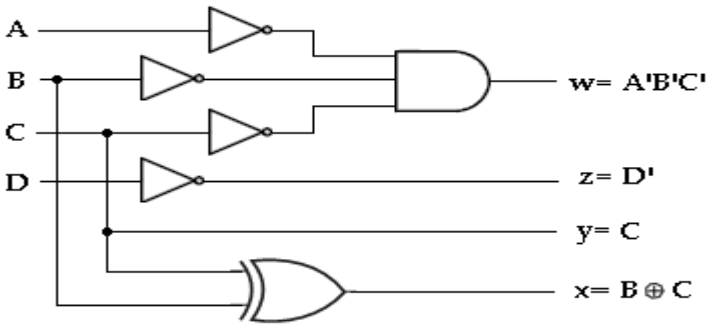
Truth Table:

Inputs				Outputs			
A	**B**	**C**	**D**	**w**	**x**	**y**	**z**
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0

K-map Simplification:



Logic Diagram:



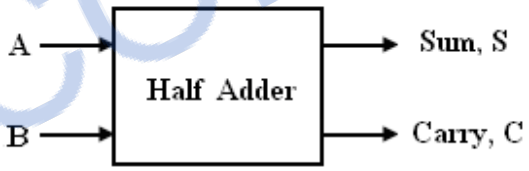
ARITHMETIC CIRCUITS:

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.



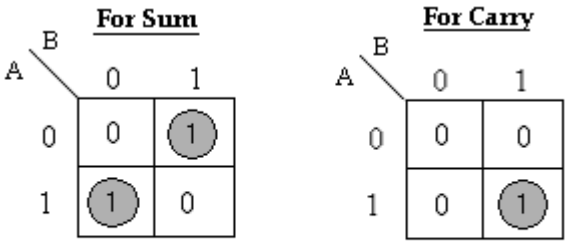
Block schematic of half-adder

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

Truth Table:

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

K-map simplification:



Sum = AB'+ A'B
= A⊕B

Carry = A .B

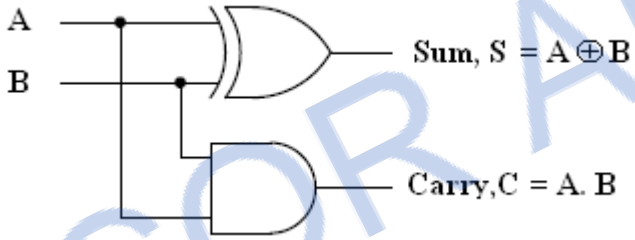
The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A'B+ AB'= A⊕B

Carry, C = A . 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,

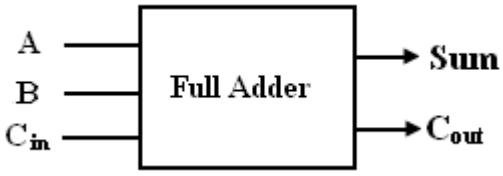


Logic Implementation of Half-adder

Full-Adder:

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

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

K-map simplification:

For Sum

A \ BC_{in}	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

Sum, S = A′B′C_{in} + A′BC′$_{in}$ + AB′C′$_{in}$ + ABC$_{in}$

For Carry

A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

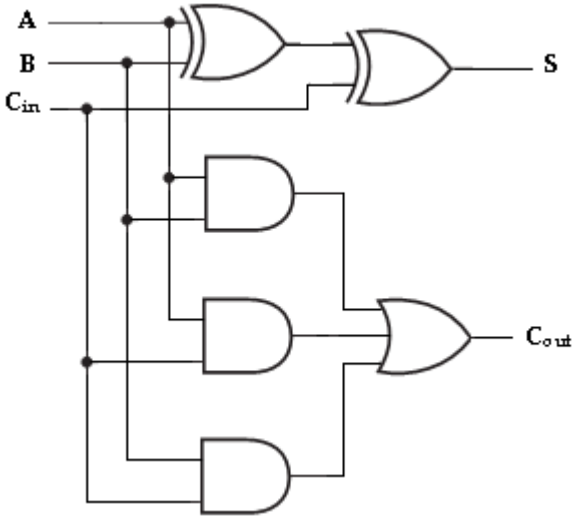
Carry, C_{out} = AB + AC_{in} + BC_{in}

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

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

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

Logic Diagram:



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

Sum = $C_{in} \oplus (A \oplus B)$ **[$x \oplus y = x'y + xy'$]**

= $C_{in} \oplus (A'B+AB')$

= $C'_{in}(A'B+AB') + C_{in}(A'B+AB')'$ **[$(x'y+xy')' = (xy+x'y')$]**

= $C'_{in}(A'B+AB') + C_{in}(AB+A'B')$

= $A'BC'_{in} + AB'C'_{in} + ABC_{in} + A'B'C_{in}$.

and the carry output is,

Carry, C_{out} = **$AB + C_{in}(A'B+AB')$**

= $AB + A'BC_{in} + AB'C_{in}$

= $B(A+A'C_{in}) + AB'C_{in}$

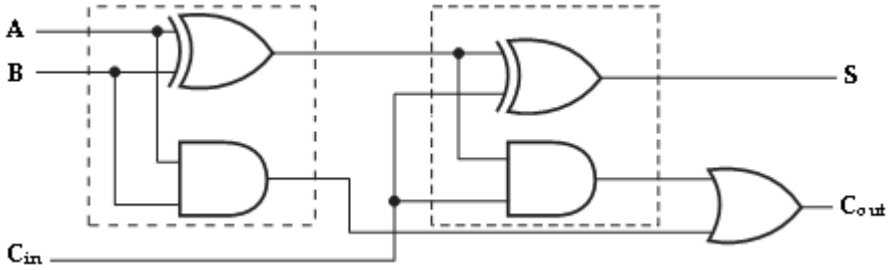
= $B(A+C_{in}) + AB'C_{in}$

= $AB + BC_{in} + AB'C_{in}$

= $AB + C_{in}(B + AB')$

= $AB + C_{in}(A + B)$

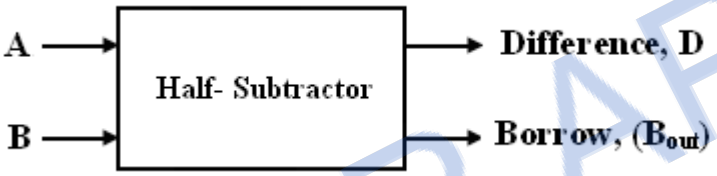
= $AB + AC_{in} + BC_{in}$.



Implementation of full adder with two half-adders and an OR gate

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.

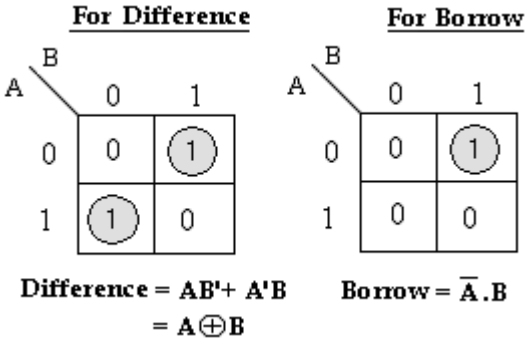


Block schematic of half-subtractor

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

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

K-map simplification:



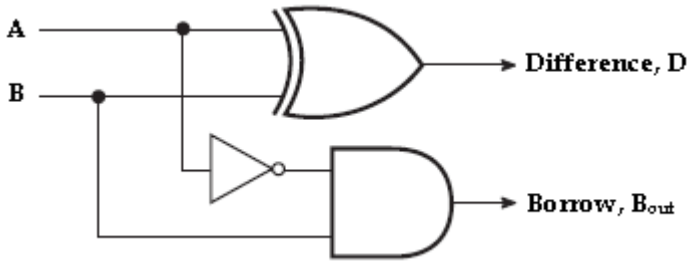
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = A'B+ AB'= A ⊕ B

Borrow, B_{out} = A' . 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 adder is,



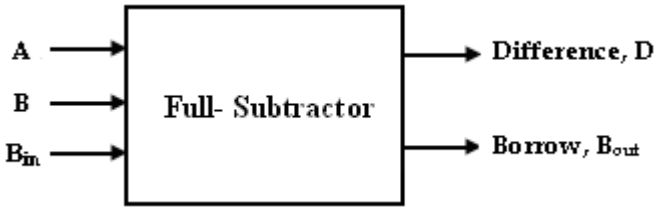
Logic Implementation of Half-Subtractor

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, ie., 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- subtractor

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 Difference

A \ BB_{in}	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

For Borrow

A \ BB_{in}	00	01	11	10
0	0	(1)	(1)	1
1	0	0	1	0

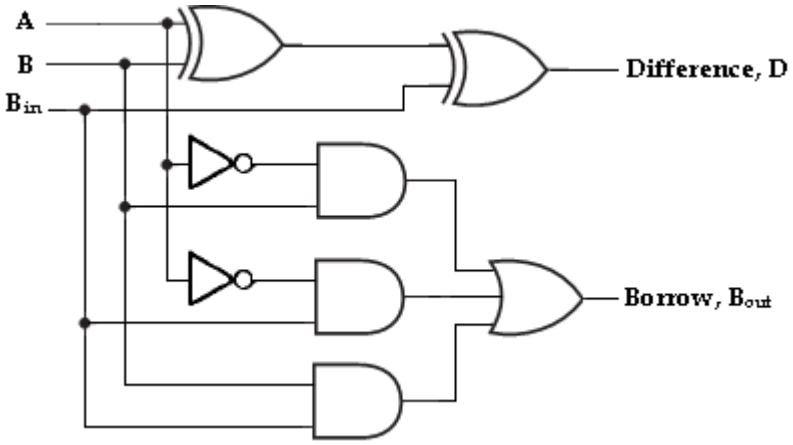
Difference, $D = A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

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

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

$$\text{Difference, } D = A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$$
$$= A'(B'B_{in} + BB'_{in}) + A(B'B'_{in} + BB_{in})$$
$$= A'(B \oplus B_{in}) + A(B \oplus B_{in})'$$
$$= A \oplus (B \oplus B_{in})$$
$$\text{Borrow, } B_{out} = A'B + A'B_{in} + BB_{in}.$$

The logic diagram for the above functions is shown as,



Implementation of full- subtractor

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

Difference,D= $B_{in} \oplus (A \oplus B)$ **[$x \oplus y = x'y + xy'$]**

$= B_{in} \oplus (A'B+AB')$

$= B'_{in}(A'B+AB') + B_{in}(A'B+AB')'$ **[$(x'y+xy')' = (xy+x'y')$]**

$= B'_{in}(A'B+AB') + B_{in}(AB+A'B')$

$= A'BB'_{in} + AB'B'_{in} + ABB_{in} + A'B'B_{in}$.

and the borrow output is,

Borrow, B_{out} = $A'B + B_{in}(A'B+AB')'$ **[$(x'y+xy')' = (xy+x'y')$]**

$= A'B + B_{in}(AB+A'B')$

$= A'B + ABB_{in} + A'B'B_{in}$

$= B(A'+AB_{in}) + A'B'B_{in}$

$= B(A'+B_{in}) + A'B'B_{in}$

$= A'B + BB_{in} + A'B'B_{in}$

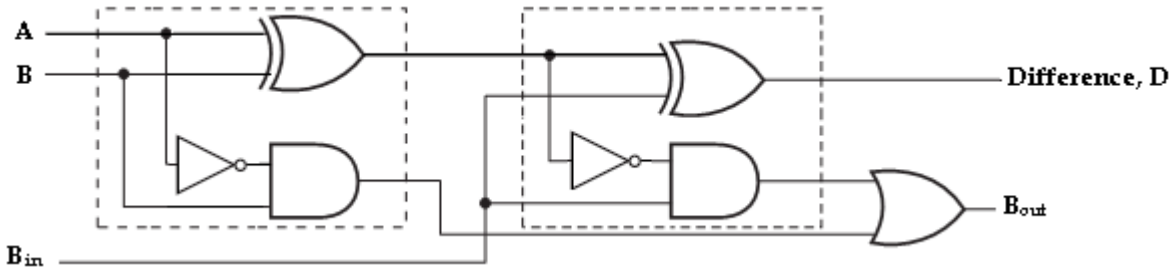
$= A'B + B_{in}(B + A'B')$

$= A'B + B_{in}(B + A')$

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

Therefore,

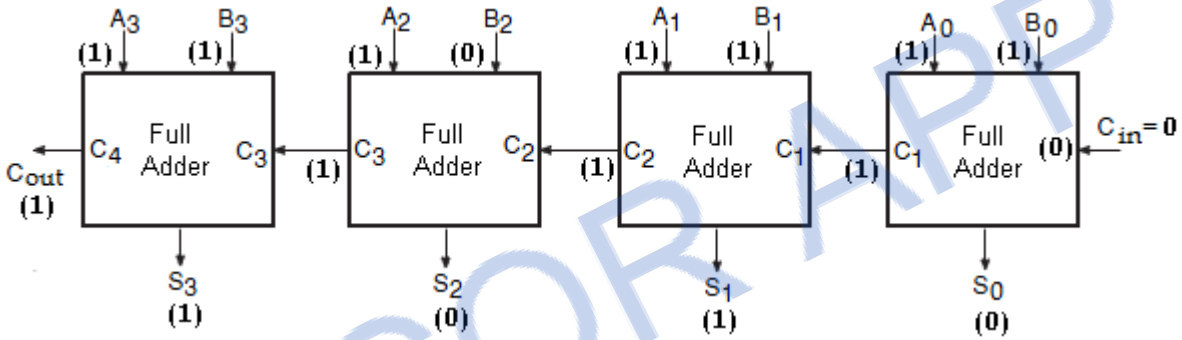
We can implement full-subtractor using two half-subtractors and OR gate as,



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,

$A_3\ A_2\ A_1\ A_0$= 1 1 1 1 and $B_3\ B_2\ B_1\ B_0$= 1 0 1 1.

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

↑ Output Carry

The bits are added with full adders, starting from the least significant position, to form the sum it 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 third 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 ($S_3 S_2 S_1 S_0$) 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.

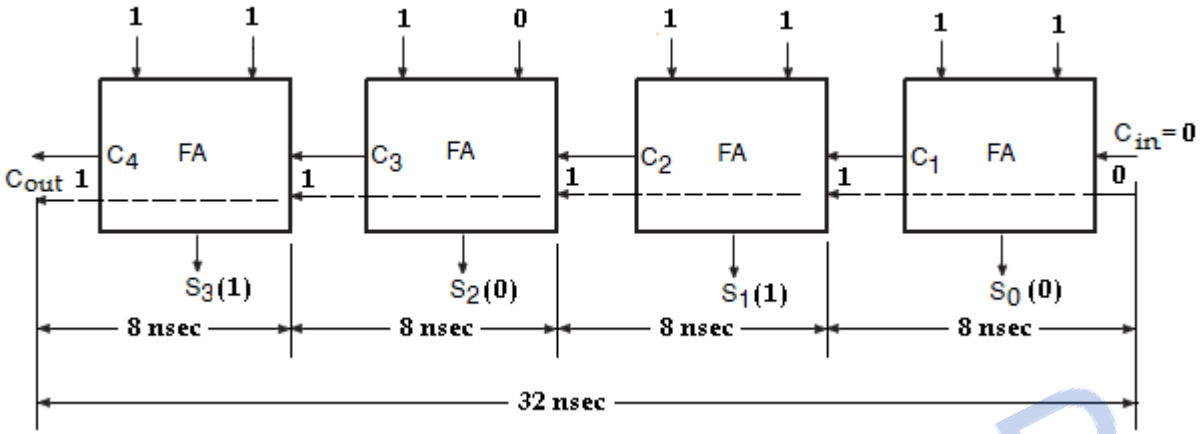
One of the methods of speeding up this process is **look-ahead carry addition** which eliminates the ripple-carry delay.

Carry Look Ahead Adder:

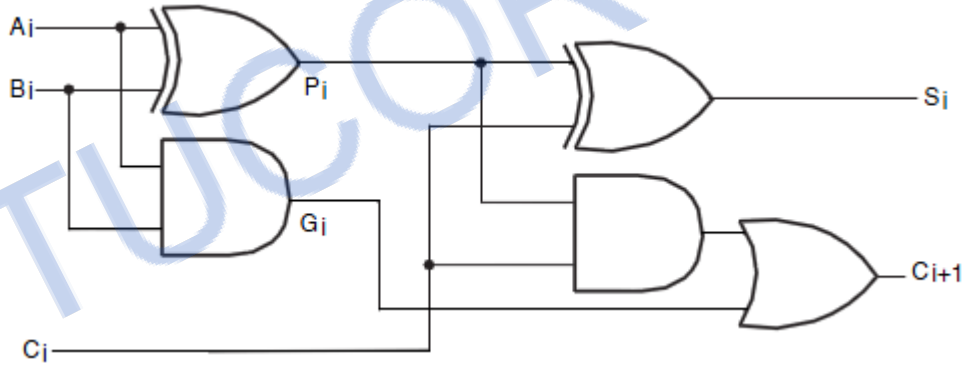
In Parallel adder, all the bits of the augend and the addend are available for computation at the same time. The carry output of each full-adder stage is connected to the carry input of the next high-order stage. Since each bit of the sum output depends on the value of the input carry, time delay occurs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

For example, addition of two numbers (1111+ 1011) gives the result as 1010. 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 8nsec, then S_3 will not react its correct value until 24 nsec after LSB is generated. Therefore total time required to perform addition is 24+ 8 = 32nsec.



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,

Carry propagate, $\mathbf{P_i = A_i \oplus B_i}$

Carry generate, $\mathbf{G_i = A_i \,.\, B_i}$

the output sum and carry can be expressed as,

$\mathbf{S_i = P_i \oplus C_i}$

$\mathbf{C_{i+1} = G_i + Pi.Ci}$

G_i (carry generate), it produces a carry 1 when both Ai and Bi are 1, regardless of the input carry C_i. Pi (carry propagate), 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= input carry

$C_1 = G_0 + P_0C_0$

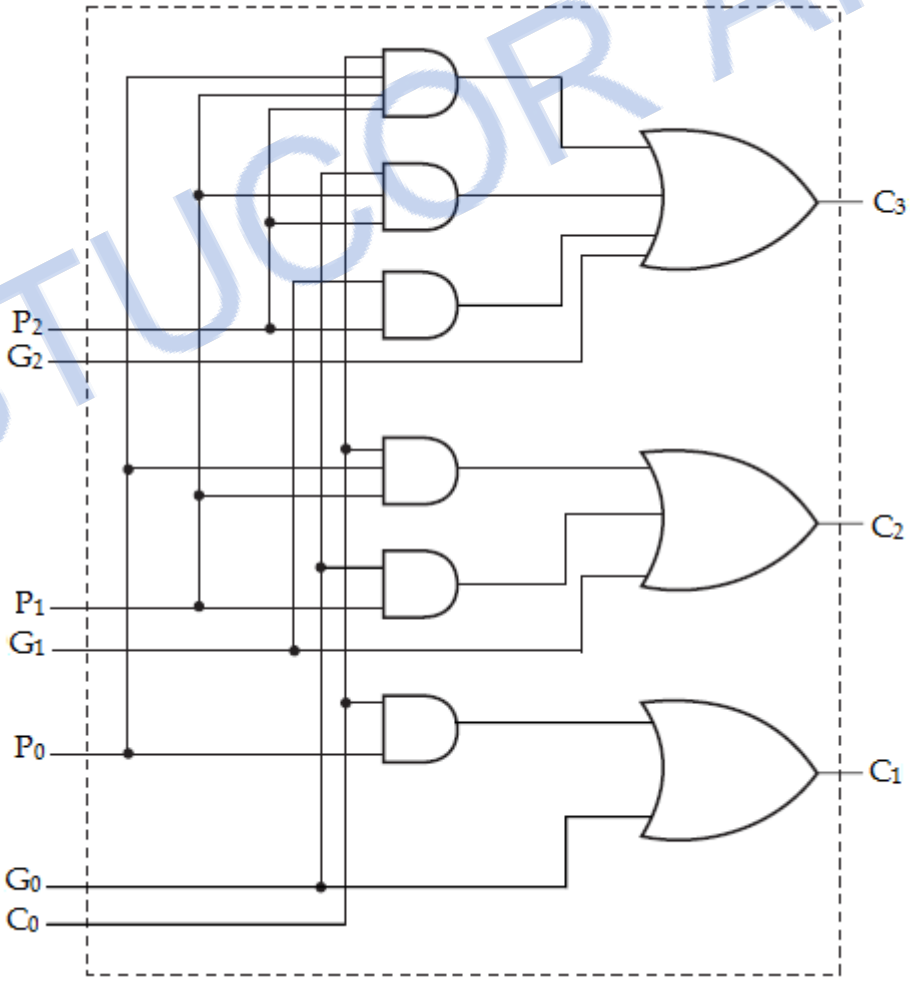
$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2 (G_1 + P_1G_0 + P_1P_0C_0)$

$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

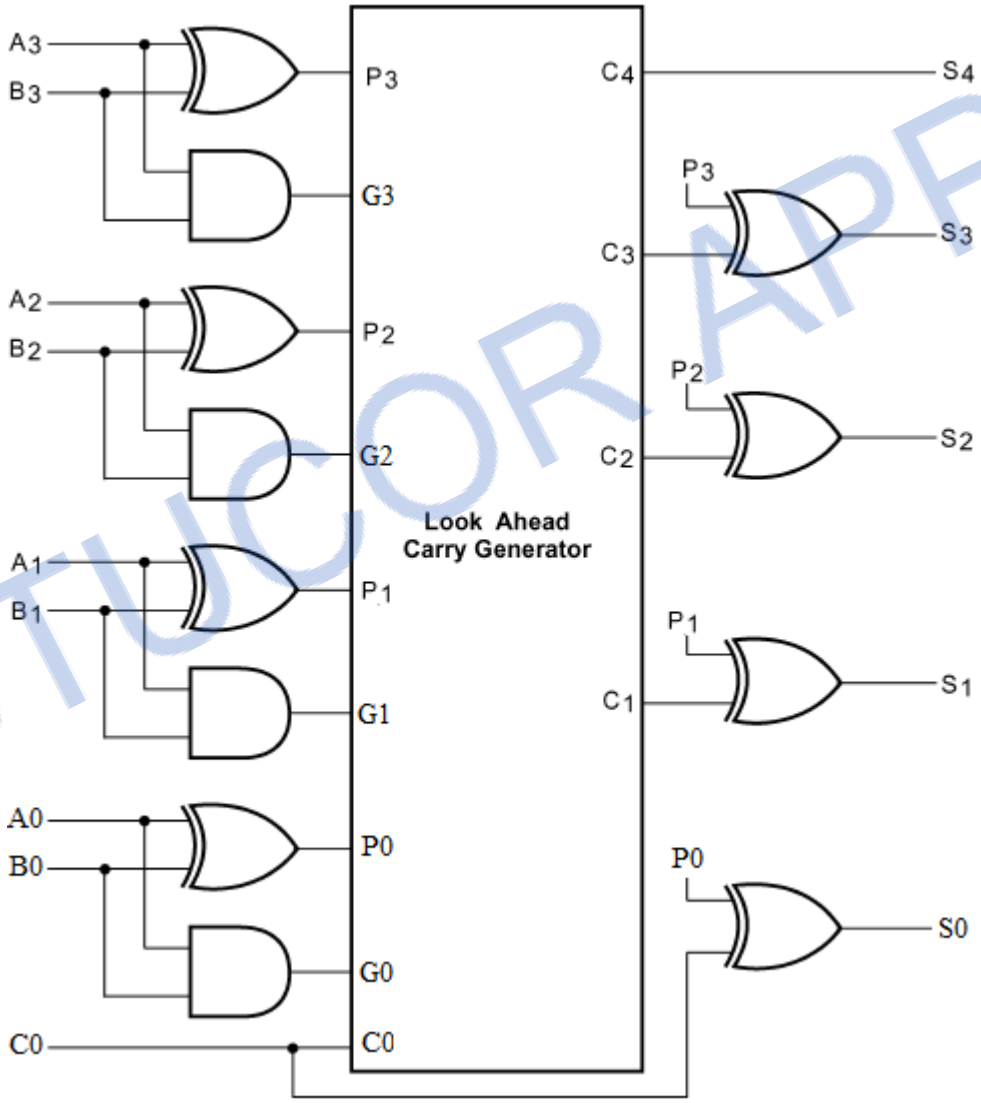
Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate. The three Boolean functions for C_1, C_2 and C_3 are implemented in the carry look-ahead generator as shown below.



Logic diagram of Carry Look-ahead Generator

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.

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.



4-Bit Adder with Carry Look-ahead

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.

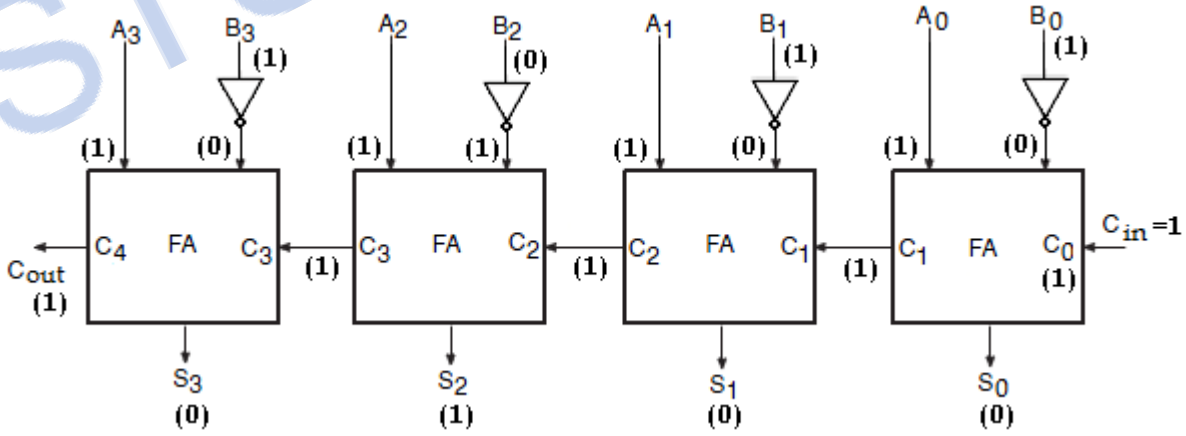
The circuit for subtracting (A – B) consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when performing subtraction. The operation thus performed becomes A, plus the 1's complement of B, plus1. This is equal to A plus the 2's complement of B.

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

$A_3\ A_2\ A_1\ A_0$= 1 1 1 1 and $B_3\ B_2\ B_1\ B_0$= 1 0 1 1.

Significant place		4	3	2	1
Minuend word A	:	1	1	1	1
Subtrahend word B	:	1	0	1	1
Input carry		1	1	1	
Minuend word A	:	1	1	1	1
1's complement of B	:	0	1	0	0
(C_{in}=1) +1	:				1
		1	0	1	0	0 ← Difference

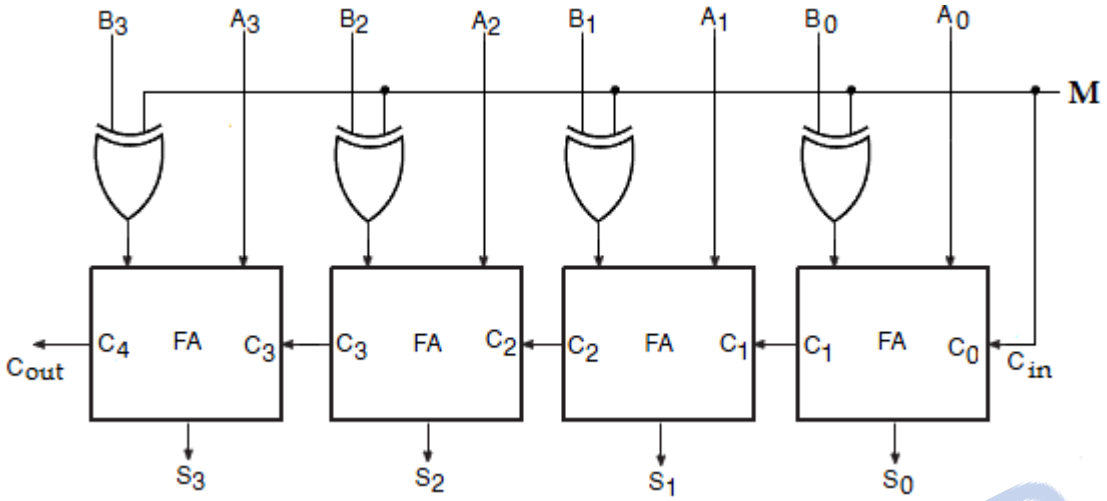
↑ Output Borrow



4-bit Parallel Subtractor

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.



4-Bit Adder Subtractor

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= 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 C, S_3, S_2, S_1, S_0, C 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.

To implement BCD adder:

- For initial addition , a 4-bit binary adder is required,
- Combinational circuit to detect whether the sum is greater than 9 and
- One more 4-bit adder to add 6 $(0110)_2$ with the sum of the first 4-bit adder, if the sum is greater than 9 or carry is 1.

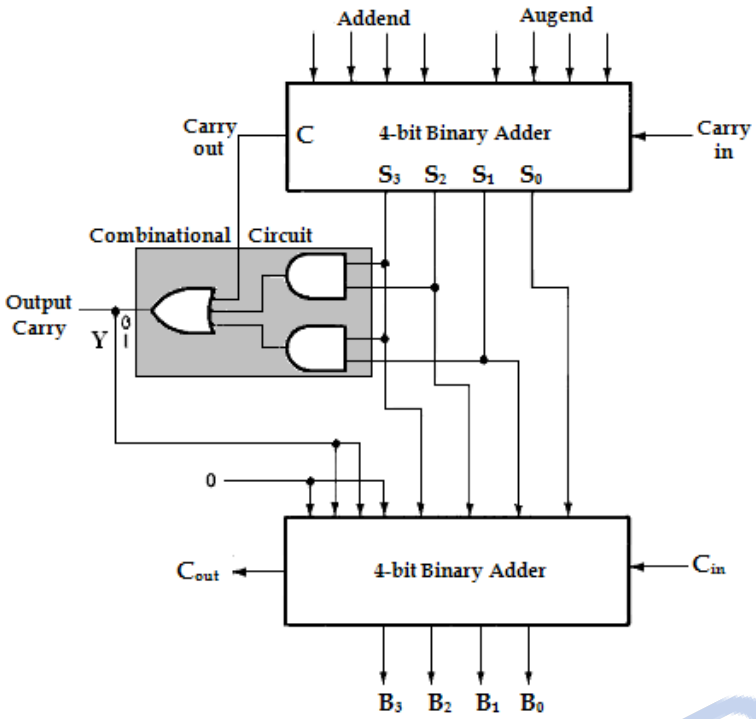
The logic circuit to detect sum greater than 9 can be determined by simplifying the Boolean expression of the given truth table.

Inputs for Combinational Circuit				Output Carry
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_3S_2 \ S_1S_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$Y = S_3S_2 + S_3S_1$

The two decimal digits, together with the input carry, are first added in the top4-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 $(0110)_2$ is added to the binary sum through the bottom 4-bit adder.

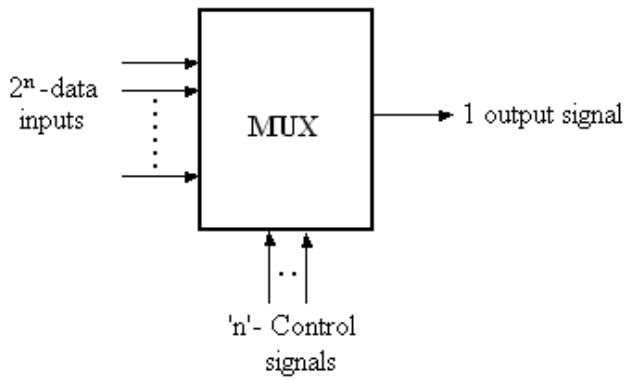


The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. The output carry from one stage must be connected to the input carry of the next higher-order stage.

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.

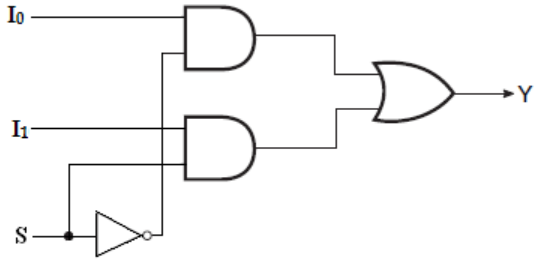
A multiplexer is also called a **data selector**, since it selects one of many inputs and steers the binary information to the output line.



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.

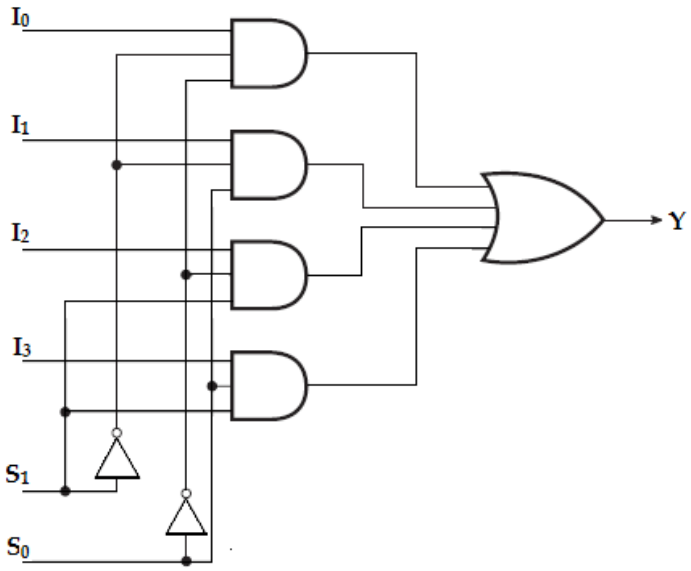
Truth table:

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 atleast 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= $I_2S_1S_0'$.

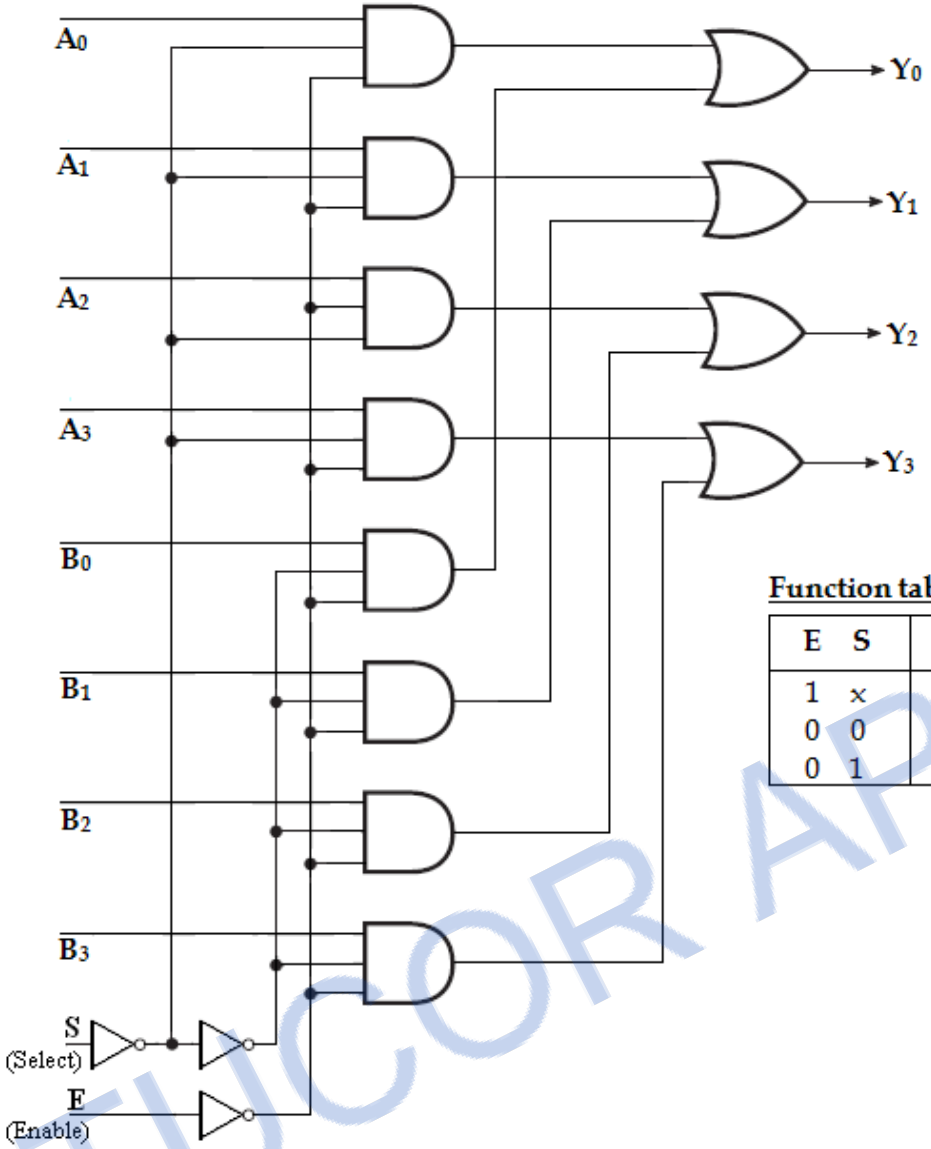
The data output is equal to I_3 only if S_1= 1 and S_0= 1; Y= $I_3S_1S_0$.

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

$$\mathbf{Y= I_0S_1'S_0'+ I_1S_1'S_0 +I_2S_1S_0'+ I_3S_1S_0.}$$

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 A0 or B0. Similarly, output Y1 may have the value of A1 or B1, 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:

1. They are used as a data selector to select out of many data inputs.
2. They can be used to implement combinational logic circuit.
3. They are used in time multiplexing systems.
4. They are used in frequency multiplexing systems.
5. They are used in A/D and D/A converter.
6. They are used in data acquisition systems.

Implementation of Boolean Function using MUX:

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

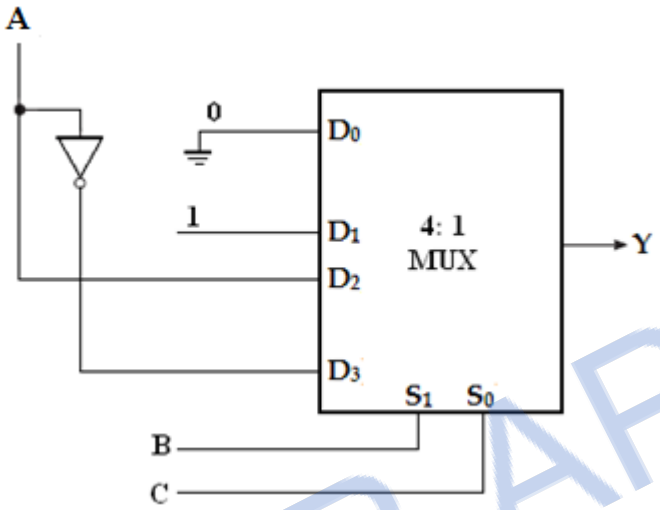
ii. List under them all the minterms in two rows as shown below.

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
$\bar{A}$	0	(1)	2	(3)
A	4	(5)	(6)	7
	0	1	A	$\bar{A}$

Multiplexer Implementation:

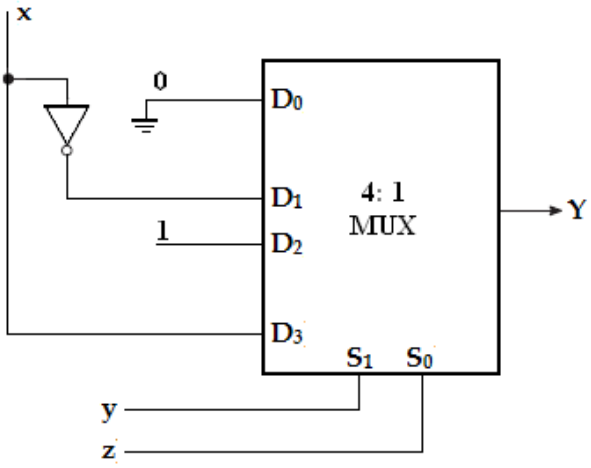


2. **F (x, y, z) = ∑m (1, 2, 6, 7)**

Implementation table:

	D_0	D_1	D_2	D_3
$\bar{x}$	0	(1)	(2)	3
x	4	5	(6)	(7)
	0	$\bar{x}$	1	x

Multiplexer Implementation:



3. **F (A, B, C) = ∑m (1, 2, 4, 5)**

Solution:

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

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

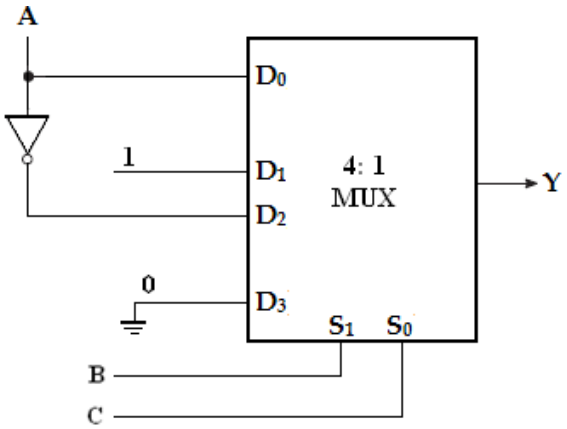
2^{n-1} to MUX i.e., 2^2 to 1 = 4 to 1 MUX

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

Implementation table:

	D_0	D_1	D_2	D_3
$\overline{A}$	0	1	2	3
A	4	5	6	7
	A	1	$\overline{A}$	0

Multiplexer Implementation



4. **F(A, B, C, D)= ∑m (0, 1, 3, 4, 8, 9, 15)**

Solution:

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

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

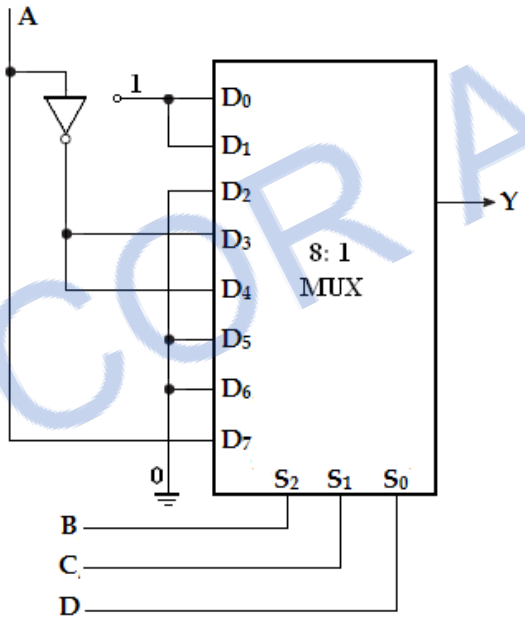
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (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
$\overline{A}$	(0)	(1)	2	(3)	(4)	5	6	7
A	(8)	(9)	10	11	12	13	14	(15)
	1	**1**	**0**	$\overline{A}$	$\overline{A}$	**0**	**0**	**A**

Multiplexer Implementation:



5. **Implement the Boolean function using 8: 1 and also using 4:1 multiplexer**

F (A, B, C, D) = ∑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)

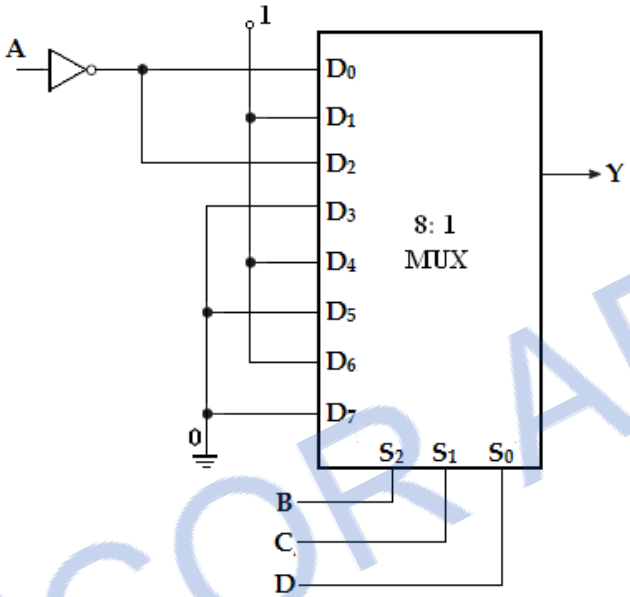
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

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

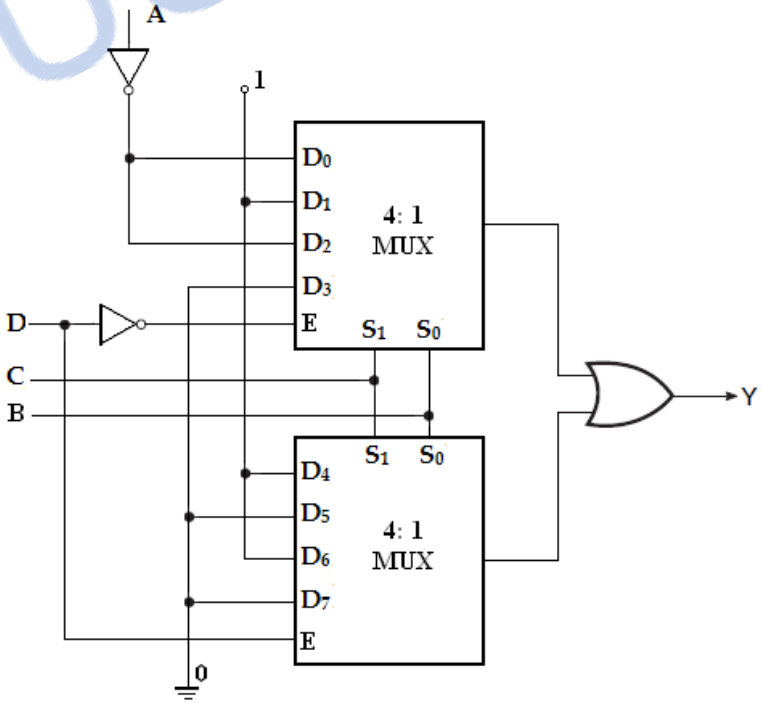
Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	(0)	(1)	(2)	3	(4)	5	(6)	7
A	8	(9)	10	11	(12)	13	(14)	15
	$\overline{A}$	**1**	$\overline{A}$	**0**	**1**	**0**	**1**	**0**

Multiplexer Implementation (Using 8: 1 MUX)



Using 4: 1 MUX:



6. **F (A, B, C, D) = ∑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)

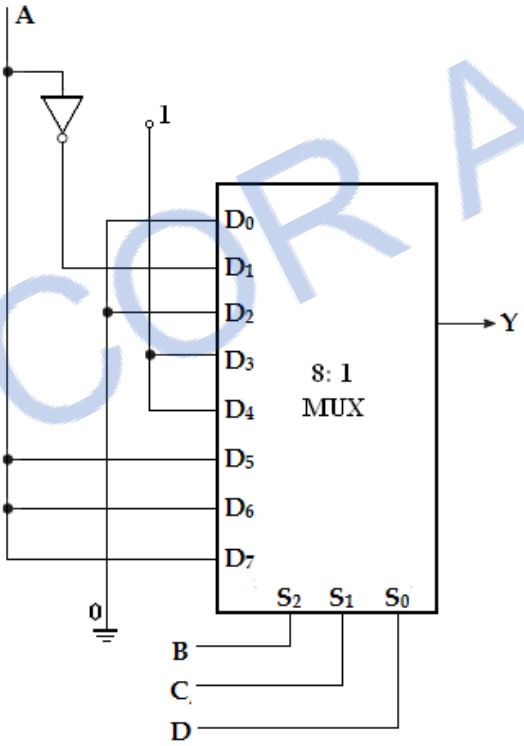
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (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
$\overline{A}$	0	(1)	2	(3)	(4)	5	6	7
A	8	9	10	(11)	(12)	(13)	(14)	(15)
	0	$\overline{A}$	**0**	**1**	**1**	**A**	**A**	**A**

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'+ AB'CD + ABCD +A'B'CD + AB'CD +A'B'C'D+ A'BC'D

= A′BC′D′ + A′BCD′+ AB′CD + ABCD +A′B′CD +A′B′C′D+ A′BC′D

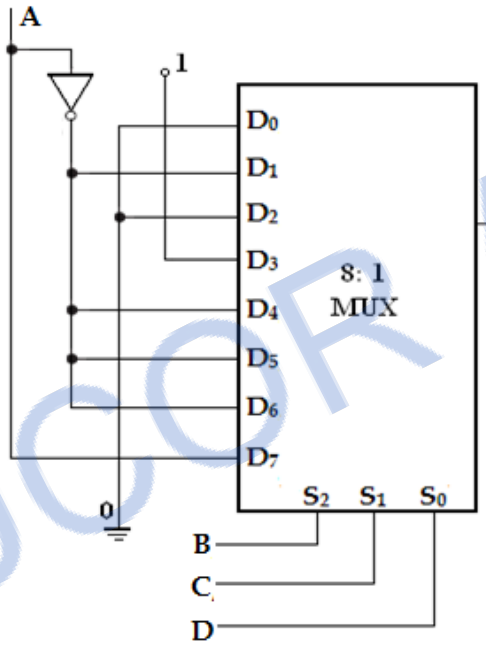
= m4+ m6+ m11+ m15+ m3+ m1+ m5

= ∑m (1, 3, 4, 5, 6, 11, 15)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	0	(1)	2	(3)	(4)	(5)	(6)	7
A	8	9	10	(11)	12	13	14	(15)
	0	$\overline{A}$	**0**	**1**	$\overline{A}$	$\overline{A}$	$\overline{A}$	A

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:

= AB′D (C′+C) + A′C′D (B′+B) + B′CD′ (A′+A) + AC′D (B′+B)

= AB′C′D + AB′CD+ A′B′C′D + A′BC′D +A′B′CD′ + AB′CD′ +AB′C′D+ ABC′D

= AB′C′D + AB′CD+ A′B′C′D + A′BC′D +A′B′CD′ + AB′CD′+ ABC′D

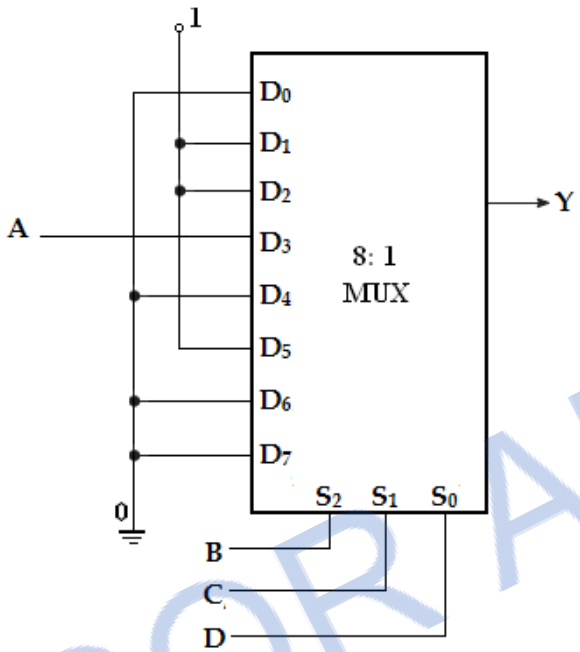
= m9+ m11+ m1+ m5+ m2+ m10+ m13

= ∑m (1, 2, 5, 9, 10, 11, 13).

Implementation Table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	0	(1)	(2)	3	4	(5)	6	7
A	8	(9)	(10)	(11)	12	(13)	14	15
	0	**1**	**1**	**A**	**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) = ∑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**)

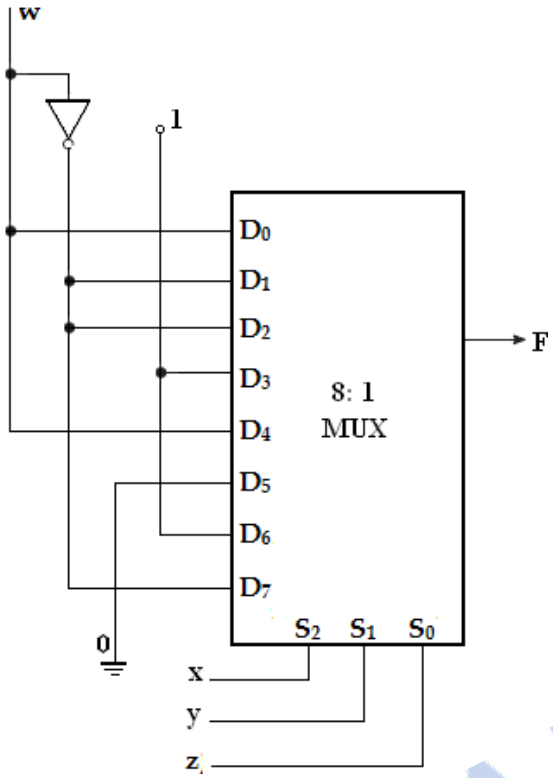
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

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

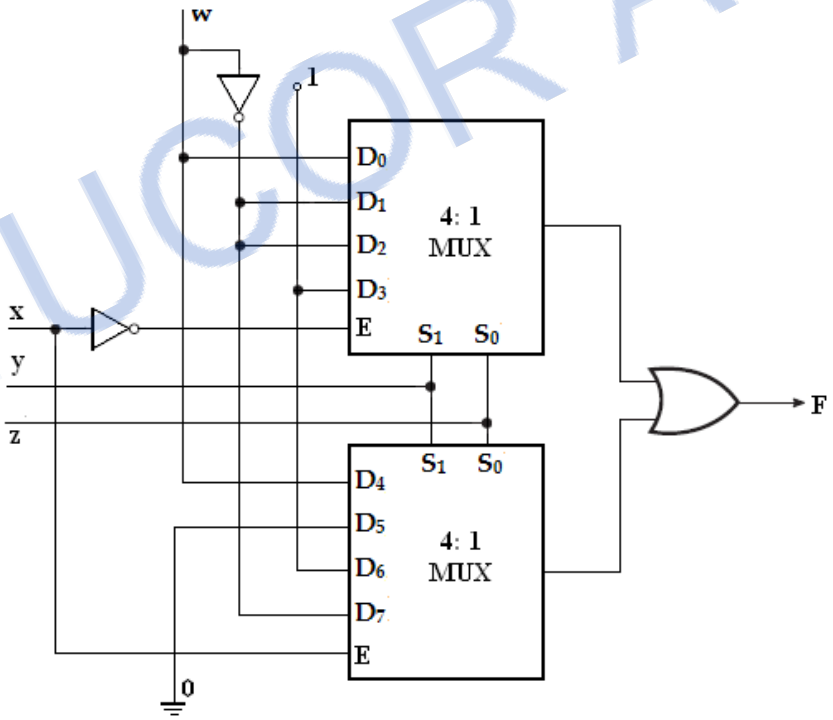
Implementation table:

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

Multiplexer Implementation (Using 8:1 MUX):



(Using 4:1 MUX)



10. Implement the Boolean function using 8: 1 multiplexer

F (A, B, C, D) = ∏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)

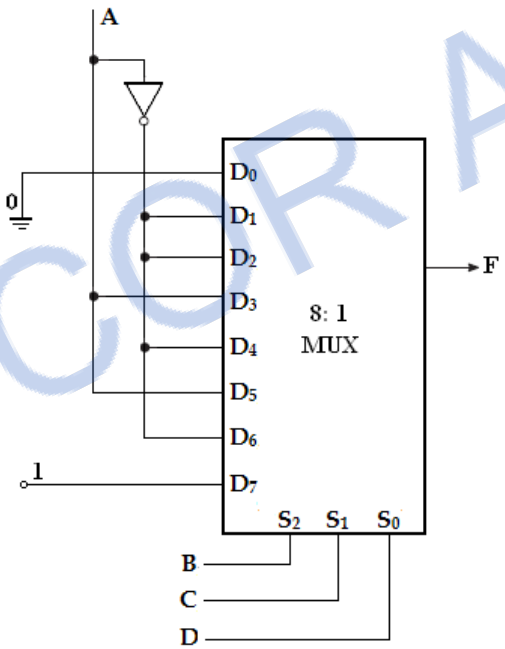
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

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
$\bar{A}$	0	(1)	(2)	3	(4)	5	(6)	(7)
A	8	9	10	(11)	12	(13)	14	(15)
	0	$\bar{A}$	$\bar{A}$	**A**	$\bar{A}$	**A**	$\bar{A}$	**1**

Multiplexer Implementation:



11. Implement the Boolean function using 8: 1 multiplexer

F (A, B, C, D) = ∑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)

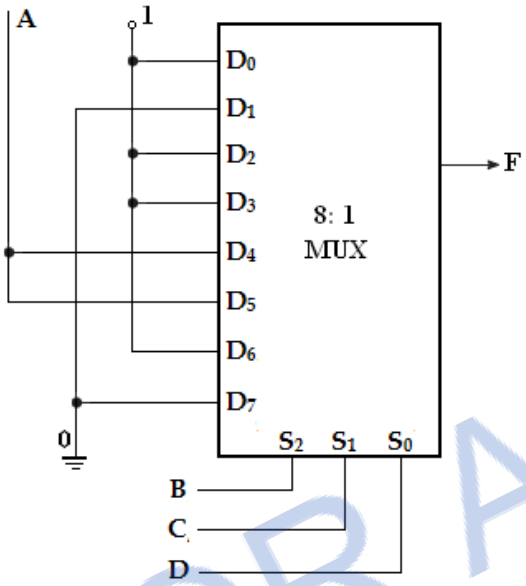
2^{n-1} to MUX i.e., 2^3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (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
$\overline{A}$	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	1	**0**	**1**	**1**	**A**	**A**	**1**	**0**

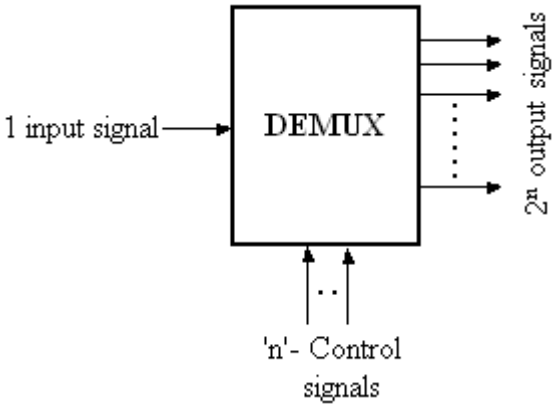
Multiplexer Implementation:



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 (2^n) output lines.

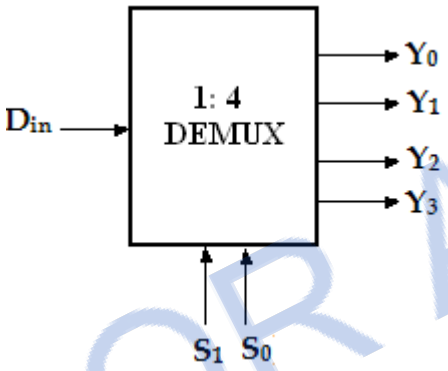


Block diagram of demultiplexer

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 distributer***" or a "***serial-to-parallel converter***".

1-to-4 line 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).



Logic Symbol

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

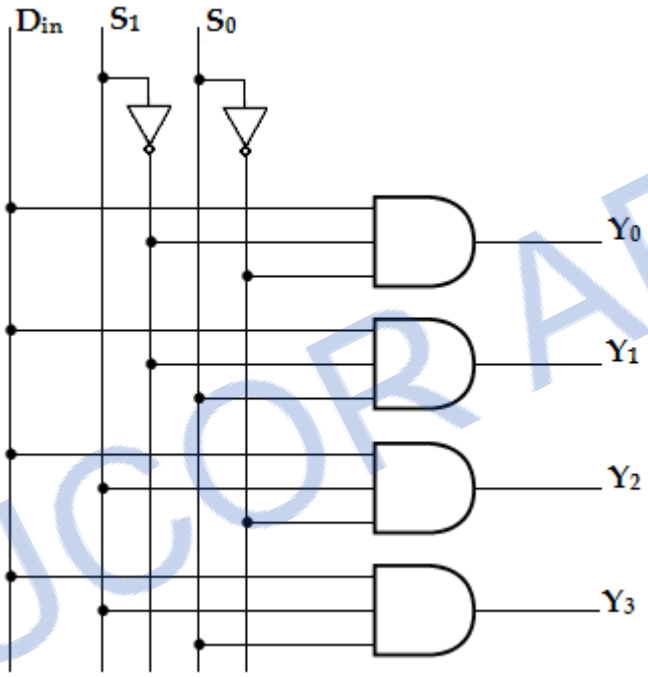
From the truth table, it is clear that the data input, D_{in} is connected to the output Y_0, when S_1= 0 and S_0= 0 and the data input is connected to output Y_1 when S_1= 0 and S_0= 1. Similarly, the data input is connected to output Y_2 and Y_3 when S_1= 1 and S_0= 0 and when S_1= 1 and S_0= 1, respectively.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

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

$$\mathbf{Y_1 = S_1'S_0D_{in}}$$

$$\mathbf{Y_2 = S_1S_0'D_{in}}$$

$$\mathbf{Y_3 = S_1S_0D_{in}}$$



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:

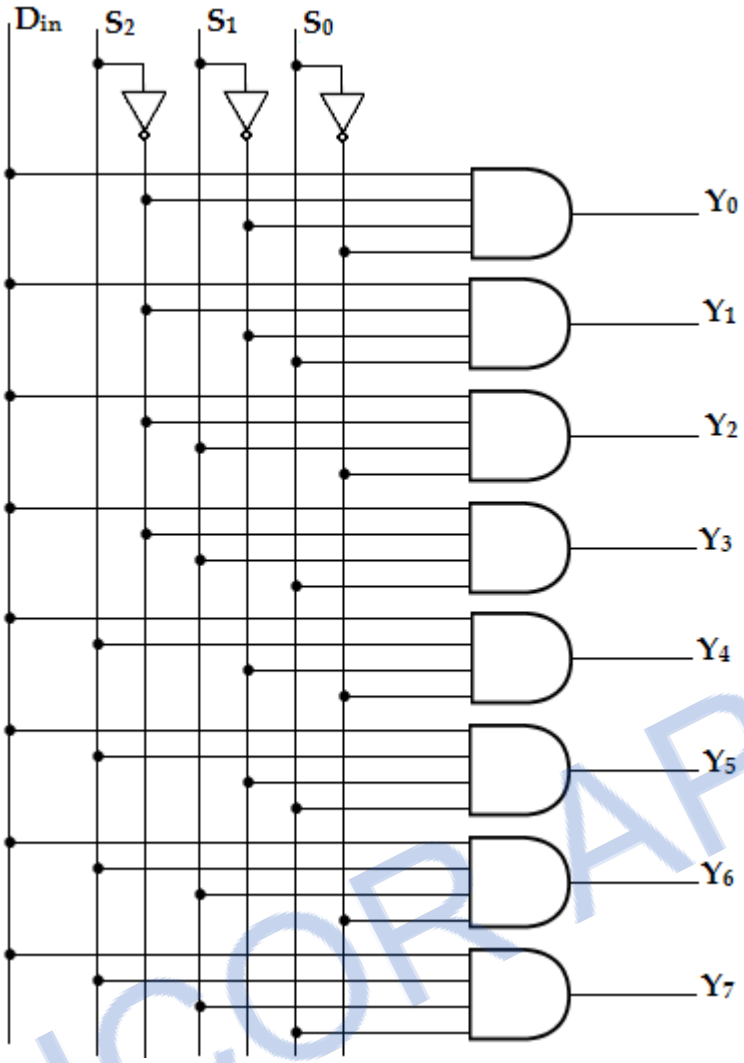
$Y_0 = S_2'S_1'S_0'D_{in}$ $\quad Y_4 = S_2 S_1'S_0'D_{in}$

$Y_1 = S_2'S_1'S_0D_{in}$ $\quad Y_5 = S_2 S_1'S_0D_{in}$

$Y_2 = S_2'S_1S_0'D_{in}$ $\quad Y_6 = S_2 S_1S_0'D_{in}$

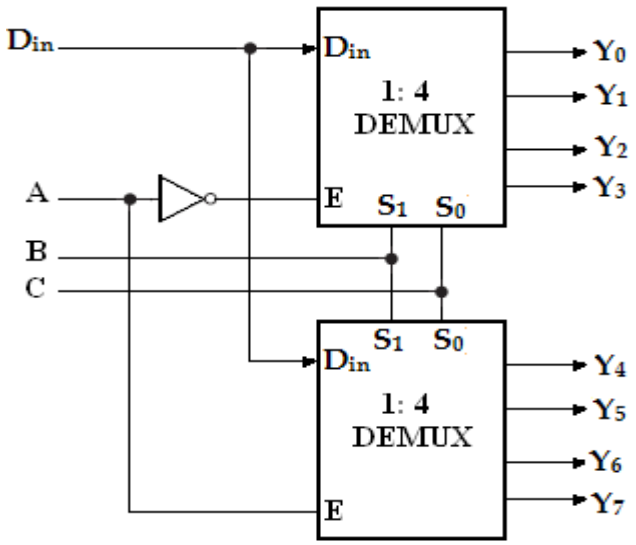
$Y_3 = S_2'S_1S_0D_{in}$ $\quad Y_7 = S_2S_1S_0D_{in}$

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 $S_2S_1S_0 = 000$, 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.



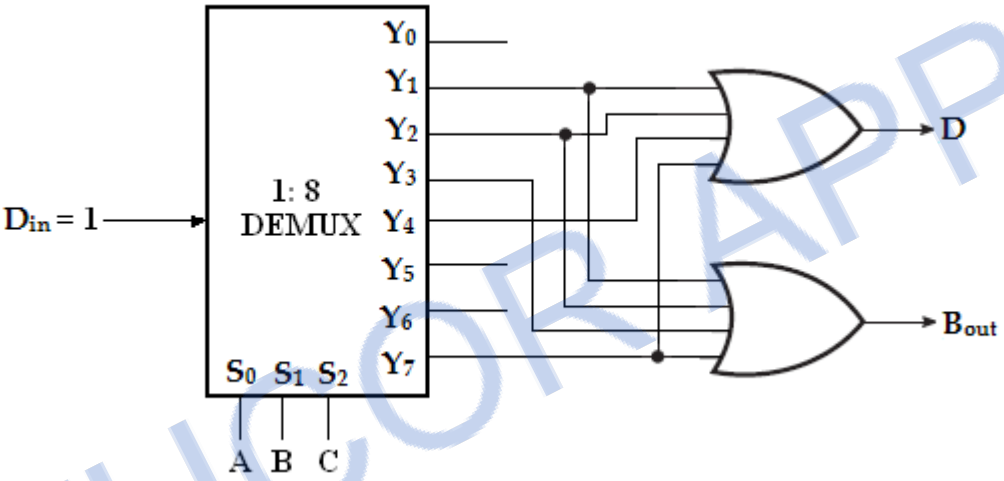
Logic diagram of 1-to-8 demultiplexer

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

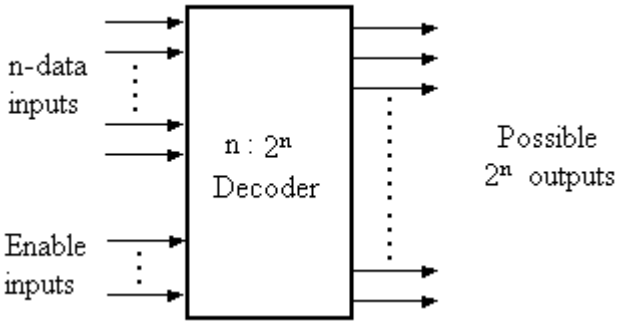


Applications:

1. It can be used as a decoder
2. It can be used as a data distributer
3. It is used in time division multiplexing at the receiving end as a data separator.
4. It can be used to implement Boolean expressions.

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

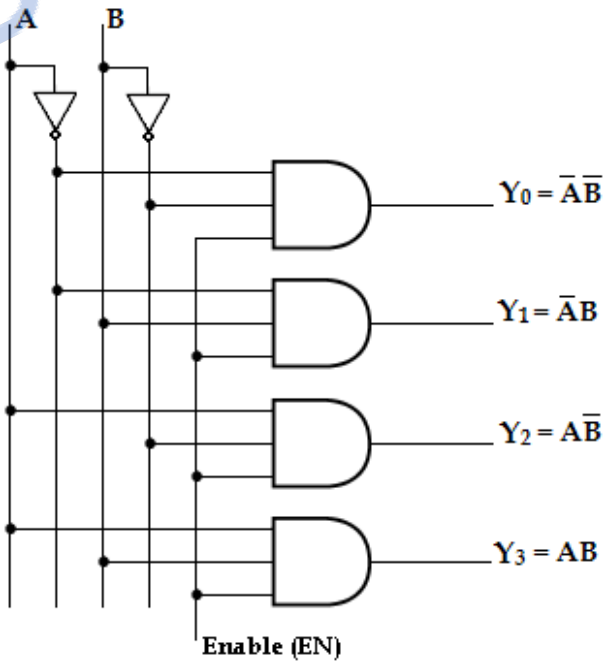


General structure of decoder

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.



2-to-4 Line decoder

Here the 2 inputs are decoded into 4 outputs, each output representing one of the minterms of the two input variables.

Inputs			Outputs			
Enable	A	B	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

As shown in the truth table, if enable input is 1 (EN= 1) only one of the outputs (Y_0 – Y_3), is active for a given input.

The output Y_0 is active, ie., Y_0= 1 when inputs A= B= 0,

Y_1 is active when inputs, A= 0 and B= 1,

Y_2 is active, when input A= 1 and B= 0,

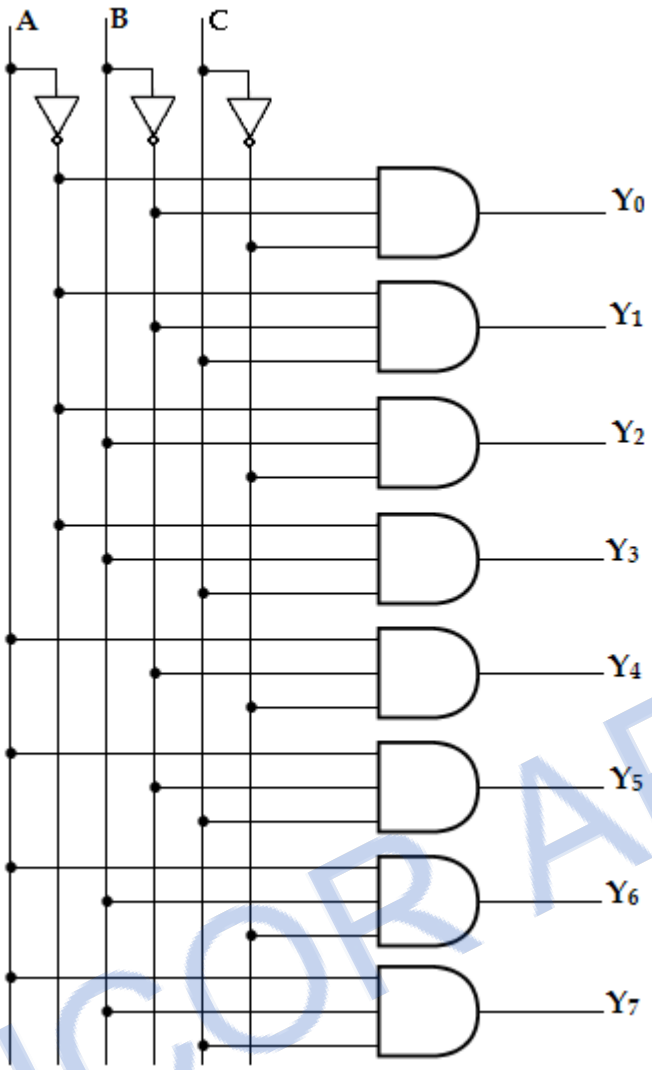
Y_3 is active, when inputs A= B= 1.

3-to-8 Line Decoder

A 3-to-8 line decoder has three inputs (A, B, C) and eight outputs (Y_0- Y_7). Based on the 3 inputs one of the eight outputs is selected.

The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. This decoder is used for binary-to-octal conversion. The input variables may represent a binary number and the outputs will represent the eight digits in the octal number system. The output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

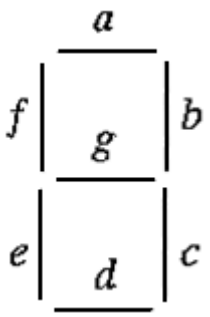
Inputs			Outputs							
A	B	C	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



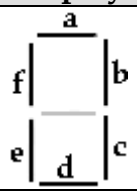
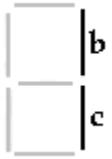
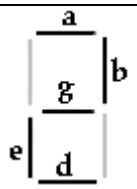
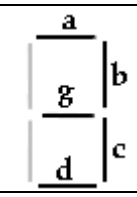
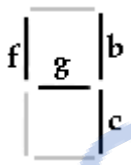
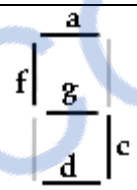
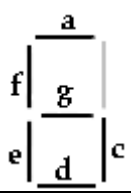
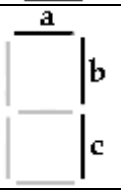
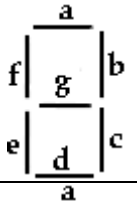
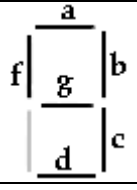
3-to-8 line decoder

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

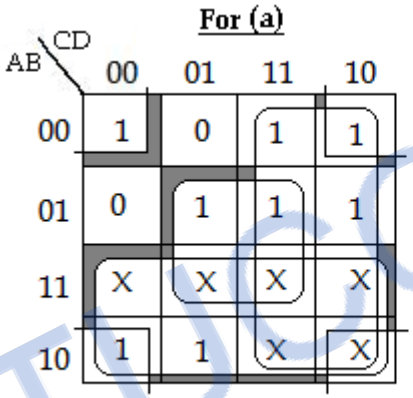
Digit	Display	Segments Activated
0		a, b, c, d, e, f
1		b, c
2		a, b, d, e, g
3		a, b, c, d, g
4		b, c, f, g
5		a, c, d, f, g
6		a, c, d, e, f, g
7		a, b, c
8		a, b, c, d, e, f, g
9		a, b, c, d, f, g

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

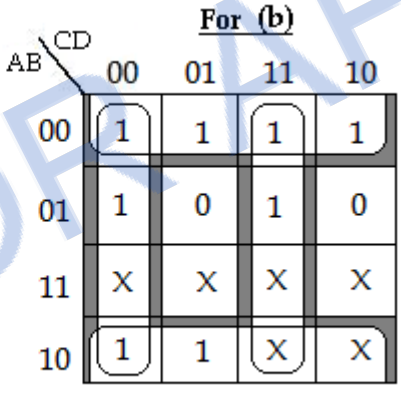
K-map Simplification:

For (a)



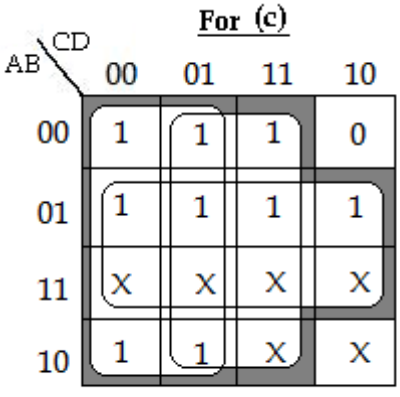
a= A+ C+ BD+ B'D'

For (b)



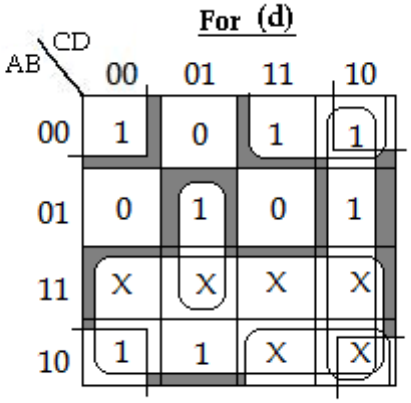
b= B'+ C'D'+ CD

For (c)



c= B+ C'+ D

For (d)



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

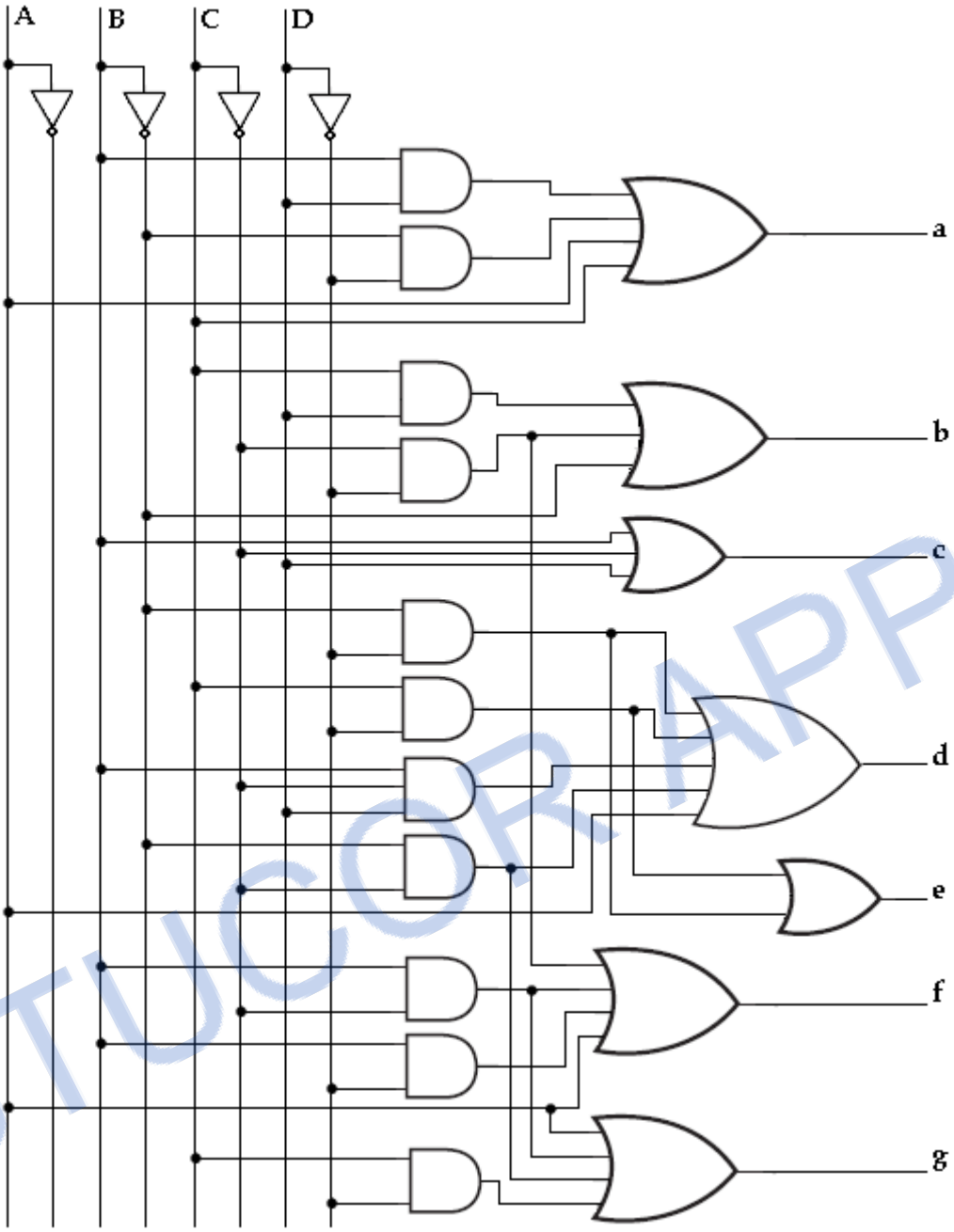
f= A+ C'D'+ BC'+ BD'

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

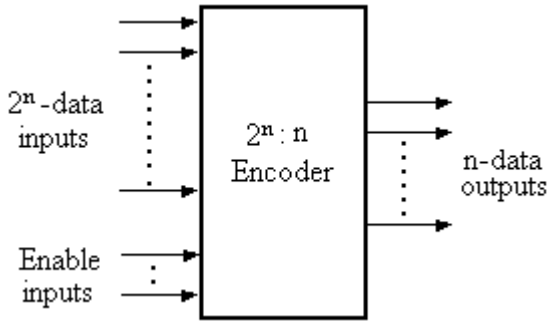
Applications of decoders:

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.
4. Address decoding
5. Implementation of combinational circuits.
6. Code converters.

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



General structure of Encoder

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.

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.

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

These conditions can be expressed by the following output Boolean functions:

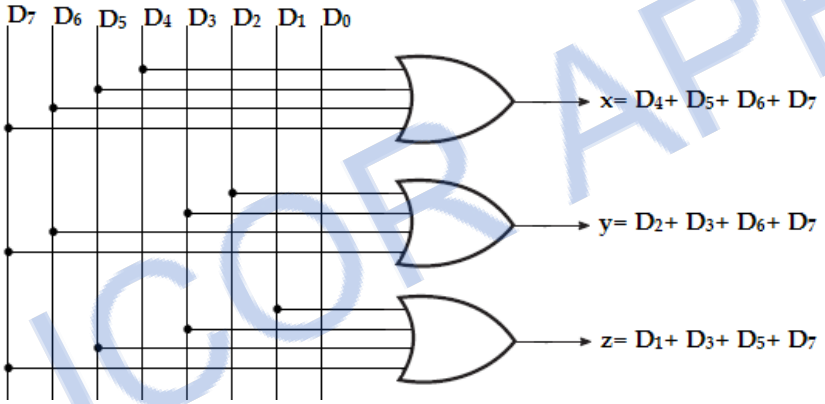
$$z = D_1 + D_3 + D_5 + D_7$$

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

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

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, 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	1 1	0 0	0 0	0	1	1
0 0 1 1	0 1 0 1	1 1 1 1	0 0 0 0	1	0	1
0 0 0 0 1 1 1 1	0 0 1 1 0 0 1 1	0 1 0 1 0 1 0 1	1 1 1 1 1 1 1 1	1	1	1

K-map Simplification:

For x

D_0D_1 \ D_2D_3	00	01	11	10
00	x	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

$x = D_2 + D_3$

For y

D_0D_1 \ D_2D_3	00	01	11	10
00	x	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

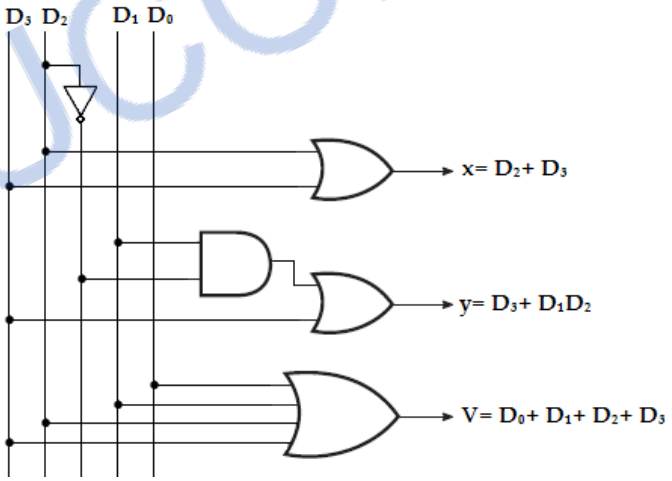
$y = D_3 + D_1D_2$

For V

D_0D_1 \ D_2D_3	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$V = D_0 + D_1 + D_2 + D_3$

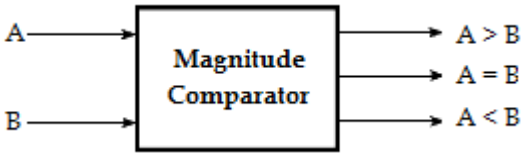
The priority encoder is implemented according to the above Boolean functions.



4-Input Priority Encoder

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.



Block diagram of magnitude comparator

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.

2.8.1 2-bit Magnitude Comparator

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

Truth table:

Inputs				Outputs		
A_1	A_0	B_1	B_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

A_1A_0 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$

For A=B

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$A=B = A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A_0B_1B_0 + A_1A_0'B_1B_0'$

$= A_1'B_1' (A_0'B_0' + A_0B_0) + A_1B_1 (A_0B_0 + A_0'B_0')$

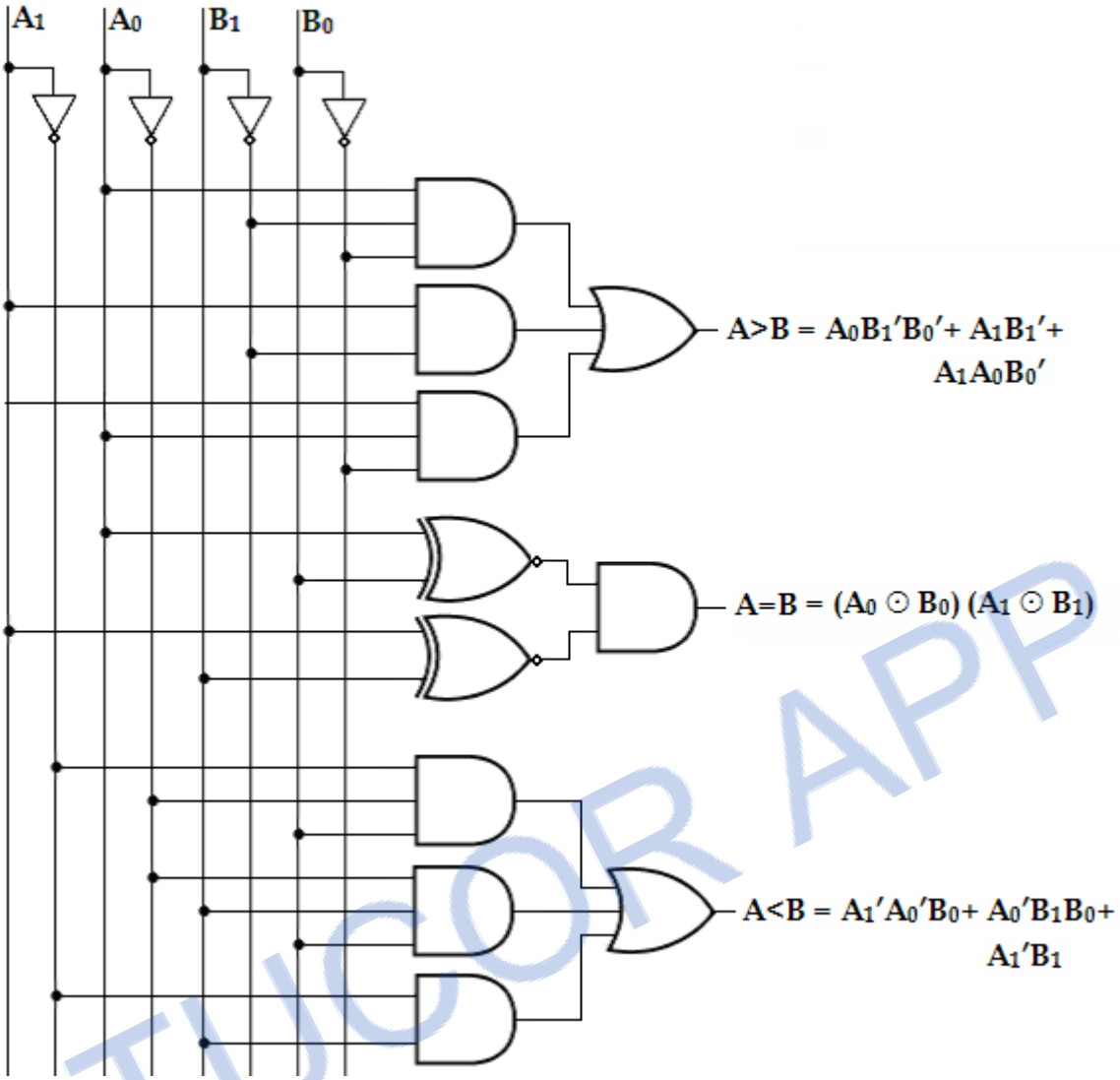
$= (A_0 \odot B_0)(A_1 \odot B_1)$

For A<B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$

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,

$\mathbf{A = A_3A_2A_1A_0}$

$\mathbf{B = B_3\,B_2\,B_1\,B_0,}$

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,

$\mathbf{Xi = AiBi + Ai'Bi'}$ for $i = 1, 2, 3, 4$.

Or, $\mathbf{Xi = (A \oplus B)'}$. or, $\mathbf{Xi' = A \oplus B}$

Or, $\mathbf{Xi = (AiBi' + Ai'Bi)'}$.

where, $Xi = 1$ 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 Xi must be equal to logic 1. This indicates the AND operation of all Xi variables. In other words, we can write the Boolean expression for two equal 4-bit numbers.

$$\mathbf{(A = B) = X_3X_2X_1 X_0.}$$

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

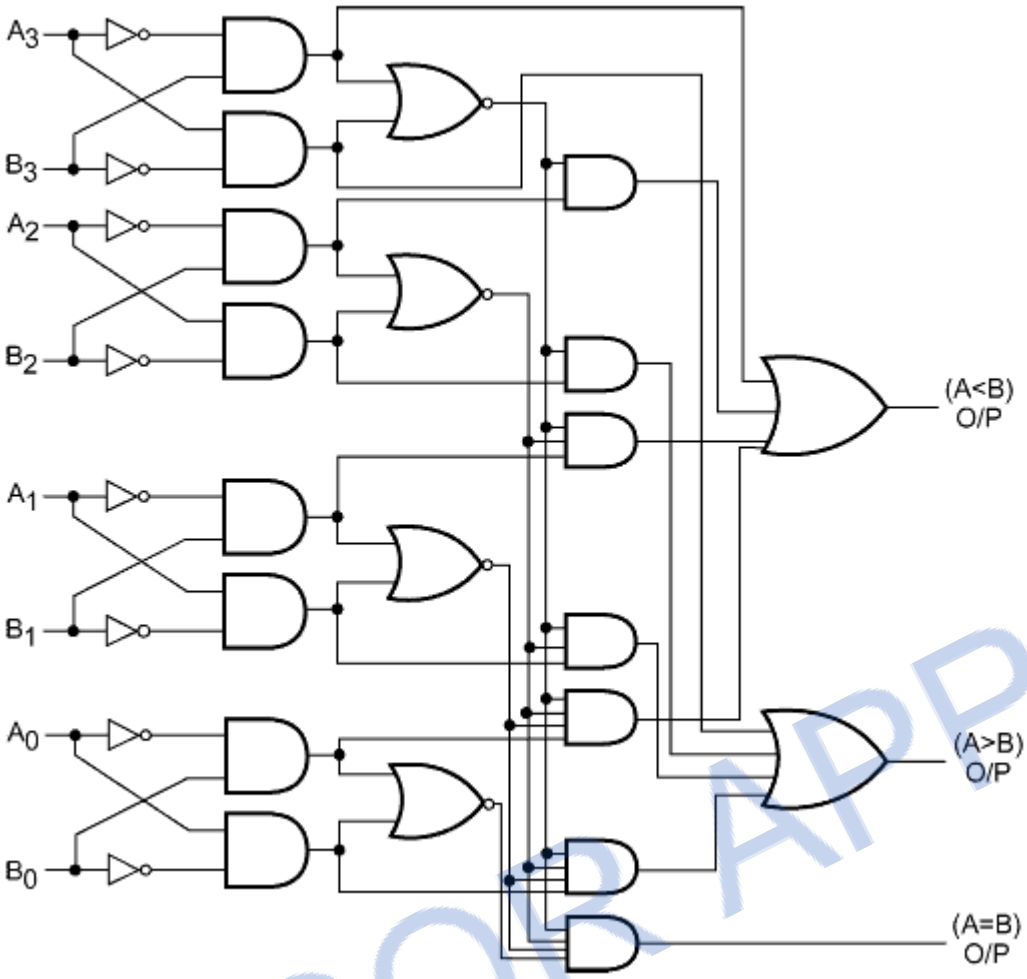
To determine if A is greater than or less than B, we inspect the relative magnitudes of pairs of significant bits starting from the most significant bit. If the two digits of the most significant position are equal, the next significant pair of digits is compared. The comparison process is continued 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,

$$\mathbf{(A>B) = A_3B_3' + X_3A_2B_2' + X_3X_2A_1B_1' + X_3X_2X_1A_0B_0'}$$

$$\mathbf{(A<B) = A_3'B_3 + X_3A_2'B_2 + X_3X_2A_1'B_1 + X_3X_2X_1A_0'B_0}$$

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.

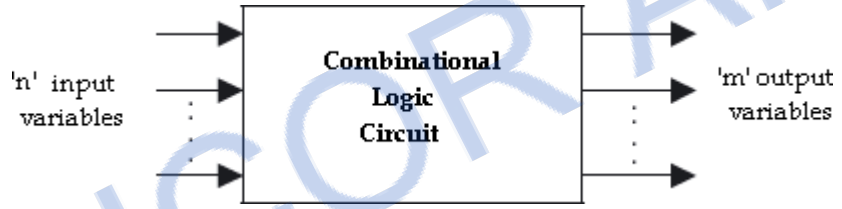
UNIT-II

SYNCHRONOUS SEQUENTIAL LOGIC

Introduction to sequential circuit- Flipflops -Operation and Excitation Tables. Triggering of FF . Analysis and design of clocked sequential circuits-Design-Moor /Mealy models, state minimization ,state assignment , circuit implementation-Registers-Counters

INTRODUCTION

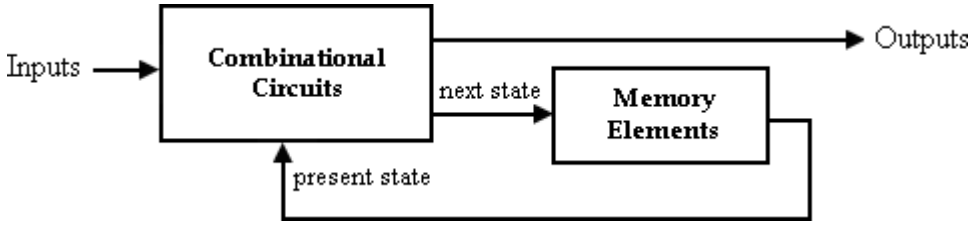
In ***combinational logic circuits***, the outputs at any instant of time depend only on the input signals present at that time. For any change in input, the output occurs immediately.



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

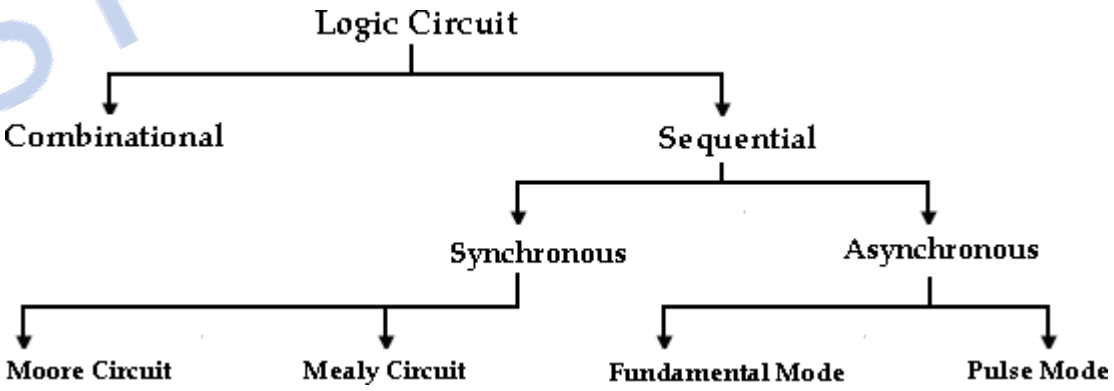
present input variables but also on the past history of input variables.

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. Adder, subtractor, Decoder, Encoders, Magnitude comparator	Eg. Shift registers, Counters

CLASSIFICATION OF LOGIC CIRCUITS



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

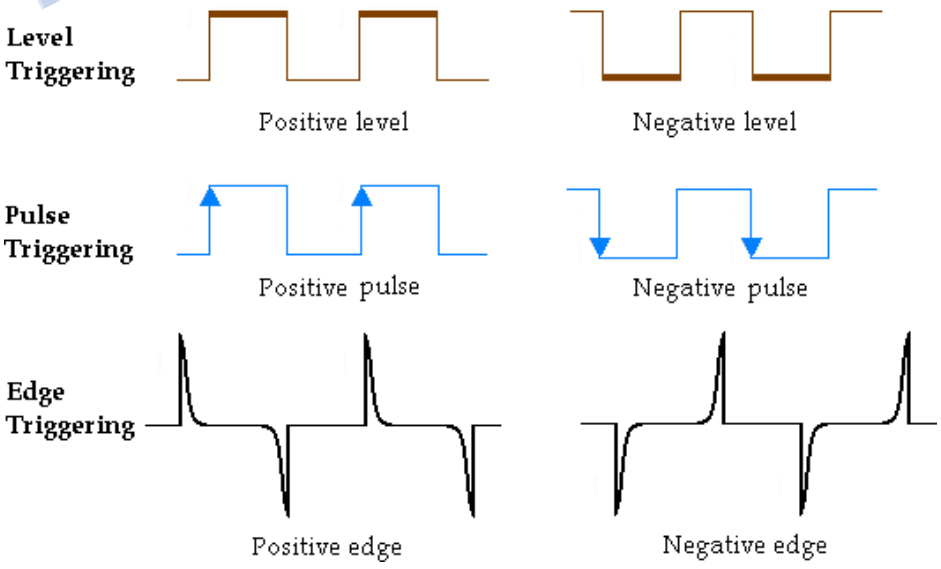
- Synchronous sequential circuits
- Asynchronous sequential circuits.

S.No	Synchronous sequential circuits	Asynchronous sequential circuits
1	Memory elements are clocked Flip-Flops.	Memory elements are either unclocked flip-flops (Latches) 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

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 1and 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.



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—

S-R Flip-Flop (**S**et – **R**eset)

J-K Flip-Flop (**J**ack **K**ilby)

D Flip-Flop (**D**elay)

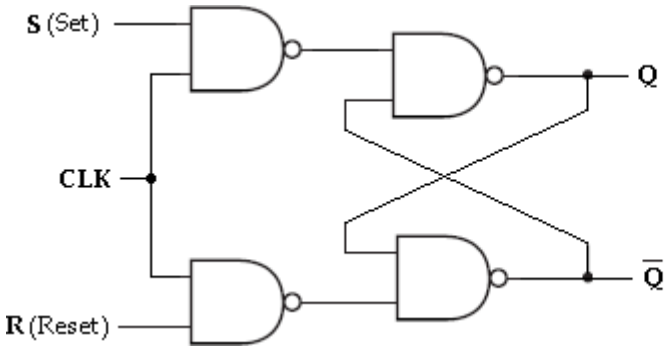
T Flip-Flop (**T**oggle)

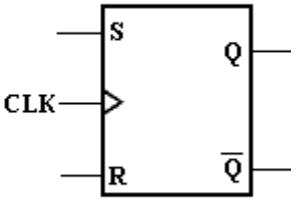
Although the S-R Flip-Flop is not available in IC form, it is the basis for the D and J-K Flip-Flops. Each type can be either positive edge-triggered (no bubble at C input) or negative edge-triggered (bubble at C input).

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**.

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.





Logic Symbol

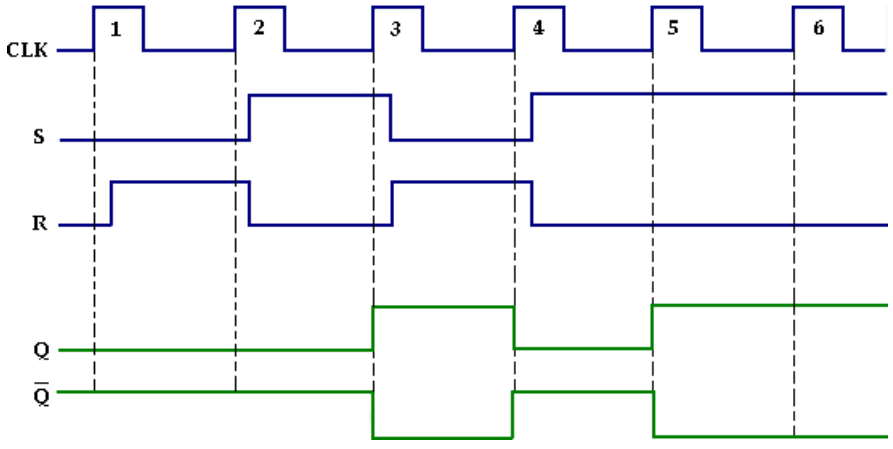
SR Flip-Flop

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	*

Truth table for SR Flip-Flop

The timing diagram of positive edge triggered SR flip-flop is shown below.



Input and output waveforms of SR Flip-Flop

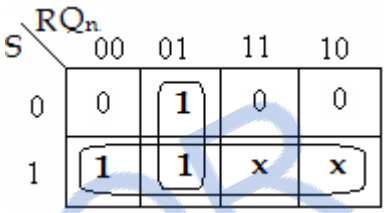
Characteristic table and Characteristic equation:

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.

S	R	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

Characteristic table

K-map Simplification:

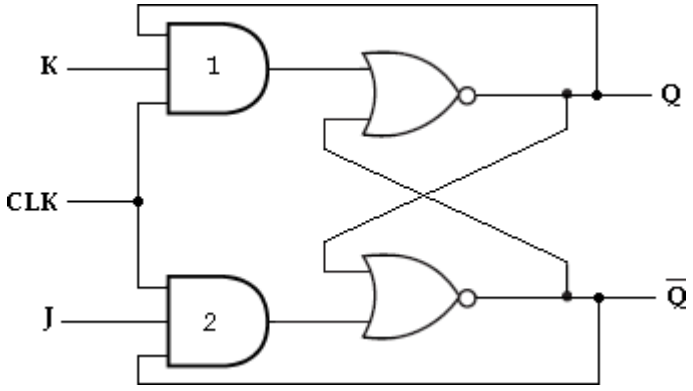


Characteristic equation:

$$Q_{n+1} = S + R'Q_n$$

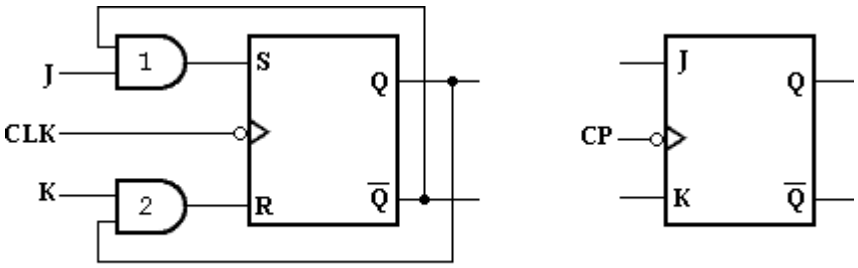
J-K Flip-Flop:

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 o 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.



(a) Using SR flipflop (b) Graphic symbol

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.

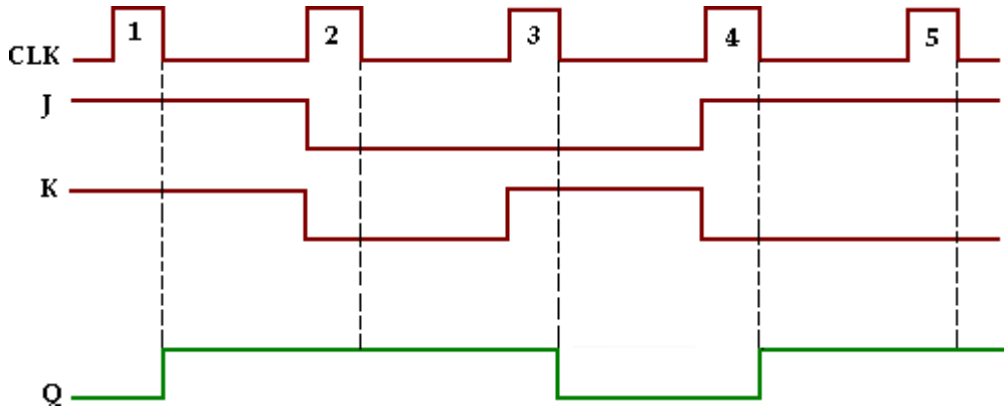
J= K= 0

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.

Truth table:

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

The timing diagram of negative edge triggered JK flip-flop is shown below.



Input and output waveforms of JK Flip-Flop

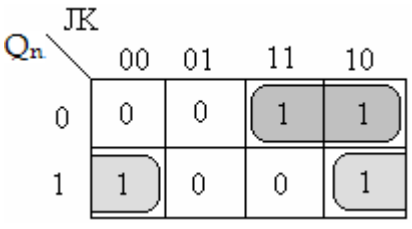
Characteristic table and Characteristic equation:

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:

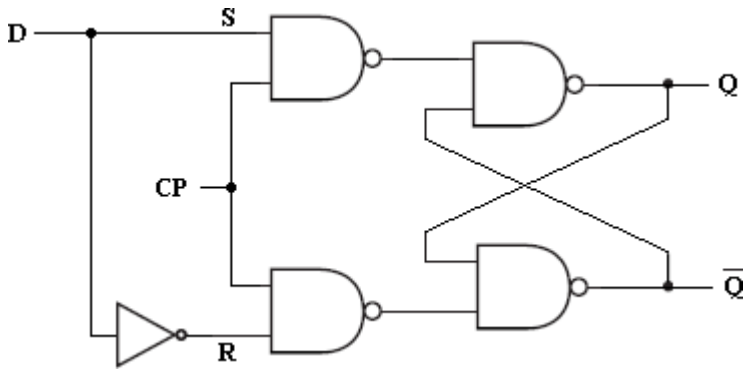


Characteristic equation:

$$Q_{n+1} = JQ' + K'Q$$

D Flip-Flop:

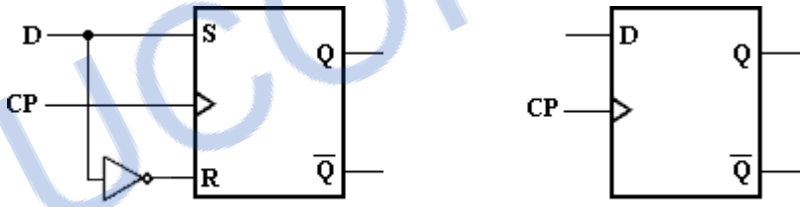
Like in D latch, in D Flip-Flop the basic SR Flip-Flop is used with complemented inputs. The D Flip-Flop is similar to D-latch except clock pulse is used instead of enable input.



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. Thisis 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

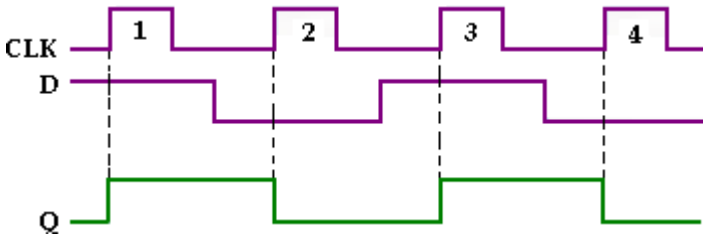
Truth Table:

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

The timing diagram of positive edge triggered D flip-flop is shown below.



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:

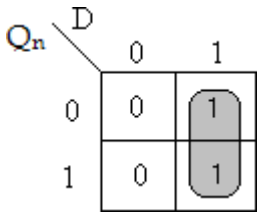
The characteristic table for D Flip-Flop shows that the next state of the Flip-Flop is independent of the present state since Q_{n+1} is equal to D. This means that an input pulse will transfer the value of input D into the output of the Flip-Flop independent of the value of the output before the pulse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

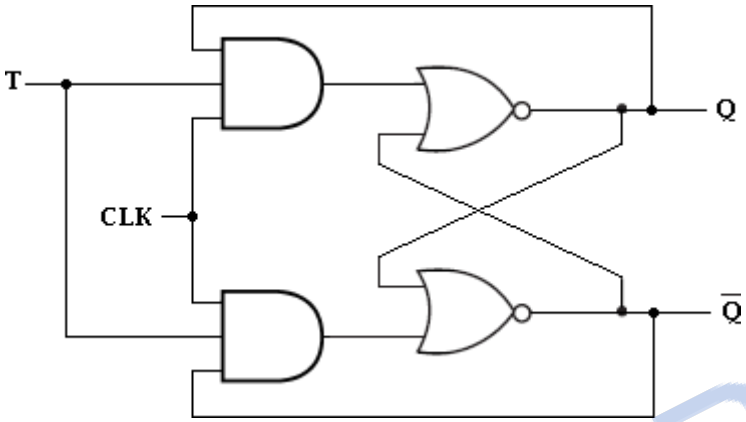


Characteristic equation:

$$Q_{n+1} = D.$$

T Flip-Flop

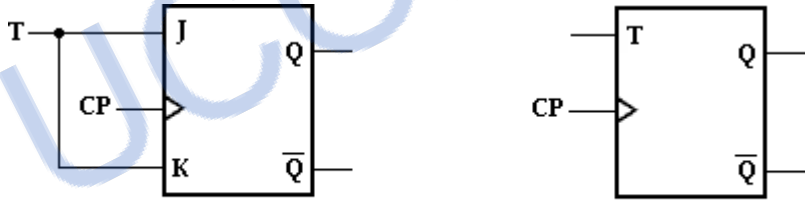
The T (*Toggle*) Flip-Flop is a modification of the JK Flip-Flop. It is obtained from JK Flip-Flop by connecting both inputs J and K together, i.e., single input. Regardless of the present state, the Flip-Flop complements its output when the clock pulse occurs while input T= 1.



T Flip-Flop

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

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



(a) Using JK flipflop (b) Graphic symbol

Truth Table:

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

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

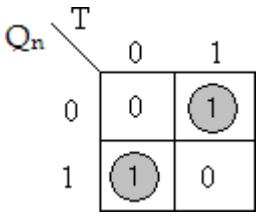
Truth table for T Flip-Flop

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:

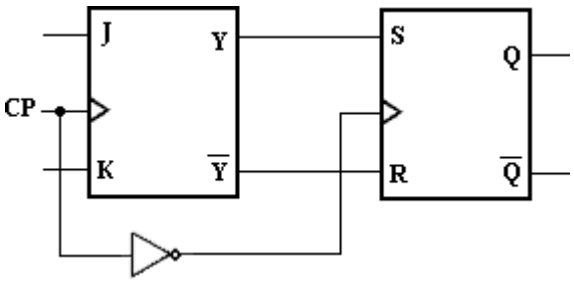


Characteristic equation:

$$Q_{n+1} = TQ_n' + T'Q_n$$

Master-Slave JK Flip-Flop

A master-slave Flip-Flop consists of clocked JK flip-flop as a master and clocked SR flip-flop as a slave. The output of the master flip-flop is fed as an input to the slave flip-flop. Clock signal is connected directly to the master flip-flop, but is connected through inverter to the slave flip-flop. Therefore, the information present at the J and K inputs is transmitted to the output of master flip-flop on the positive clock pulse and it is held there until the negative clock pulse occurs, after which it is allowed to pass through to the output of slave flip-flop. The output of the slave flip-flop is connected as a third input of the master JK flip-flop.



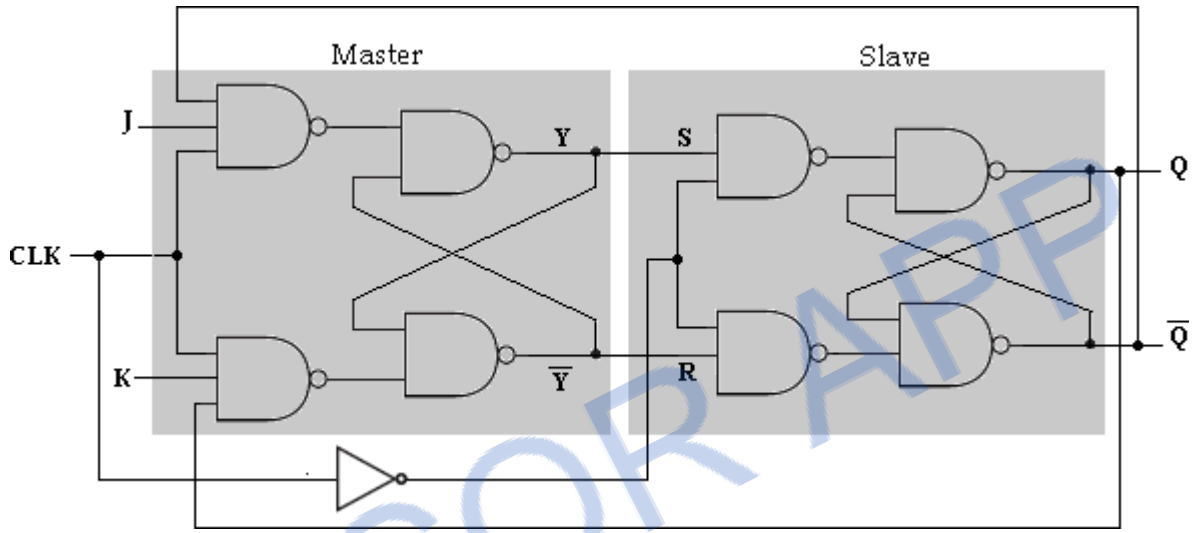
Logic diagram

When **J= 1 and K= 0**, the master sets on the positive clock. The high Y output of the master drives the S input of the slave, so at negative clock, slave sets, copying the action of the master.

When **J= 0 and K= 1**, the master resets on the positive clock. The high Y' output of the master goes to the R input of the slave. Therefore, at the negative clock slave resets, again copying the action of the master.

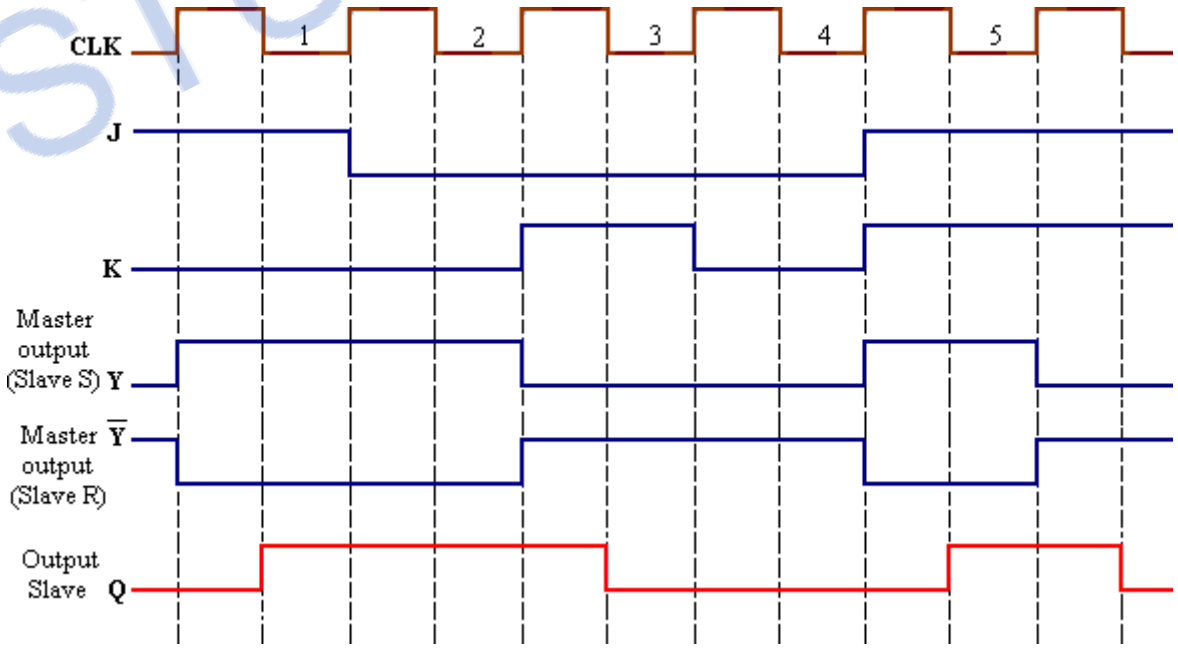
When **J= 1 and K= 1**, master toggles on the positive clock and the output of master is copied by the slave on the negative clock. At this instant, feedback inputs to the master flip-flop are complemented, but as it is negative half of the clock pulse, master flip-flop is inactive. This prevents **race around condition**.

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



Master-Slave JK Flip-Flop

The input and output waveforms of master-slave JK flip-flop is shown below.



Input and output waveform of master-slave flip-flop

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.

SR Flip- Flop:

Present State	Inputs		Next State
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	
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	
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.

JK Flip-Flop:

Present State	Inputs		Next State
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	
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	
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

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

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

Excitation Table

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

SR Flip-Flop to D Flip-Flop:

- Write the characteristic table for required Flip-Flop (D Flip-Flop).
- Write the excitation table for given Flip-Flop (SR Flip-Flop).
- Determine the expression for given Flip-Flop inputs (S & R) by using K- map.
- 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

For S

D \ Q_n	0	1
0	0	0
1	1	x

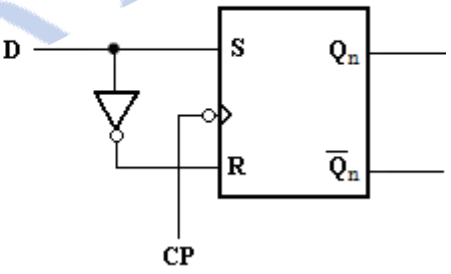
S = D

For R

D \ Q_n	0	1
0	x	1
1	0	0

R = $\overline{D}$

Logic diagram



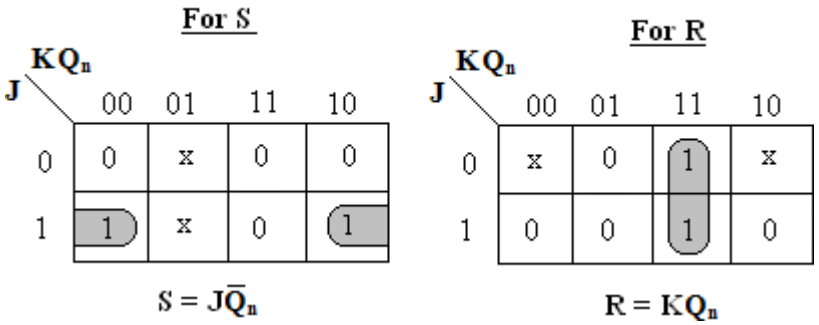
SR to D Flip-Flop

SR Flip-Flop to JK Flip-Flop

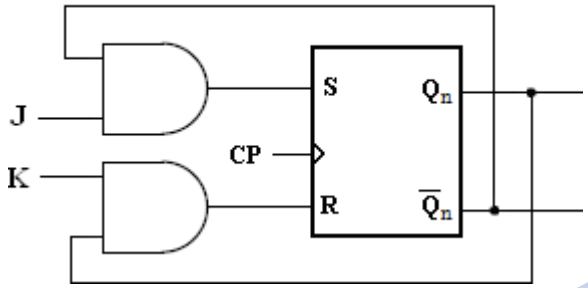
The excitation table for the above conversion is, Q_n

Inputs		Present state	Next state	Flip-Flop Inputs	
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



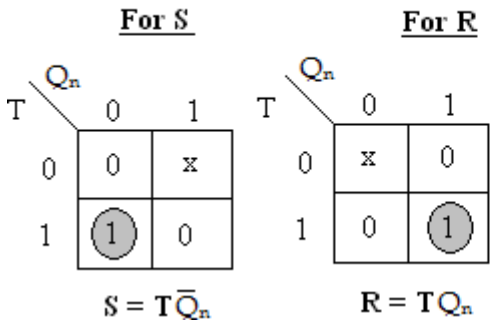
SR to JK Flip-Flop

SR Flip-Flop to T Flip-Flop

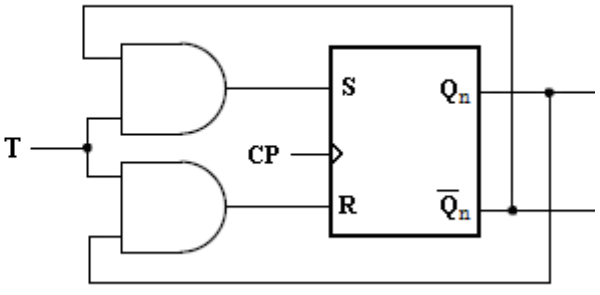
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**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



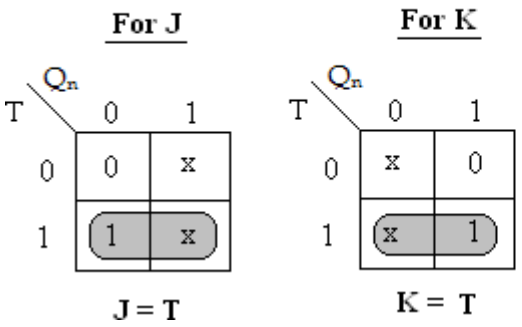
SR to T Flip-Flop

JK Flip-Flop to T Flip-Flop

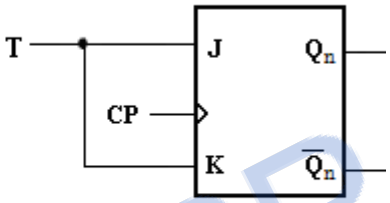
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**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



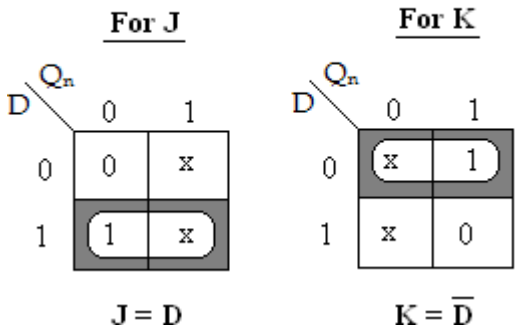
JK to T Flip-Flop

JK Flip-Flop to D Flip-Flop

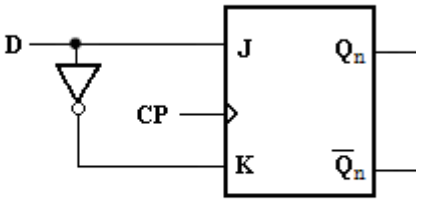
The excitation table for the above conversion is

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



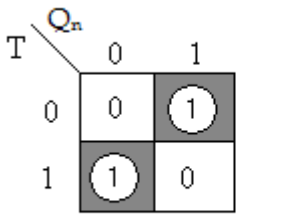
JK to D Flip-Flop

D Flip-Flop to T Flip-Flop

The excitation table for the above conversion is

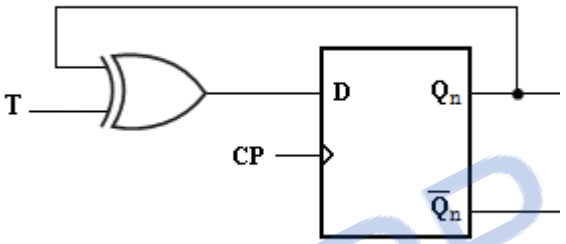
Input	Present state	Next state	Flip-Flop Input
T	**Q_n**	**Q_{n+1}**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

K-map simplification



$$D = \overline{T}Q_n + T\overline{Q}_n$$
$$= T \oplus Q_n$$

Logic diagram



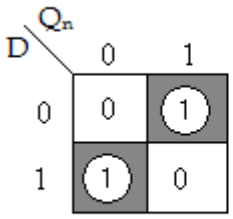
D to T Flip-Flop

T Flip-Flop to D Flip-Flop

The excitation table for the above conversion is

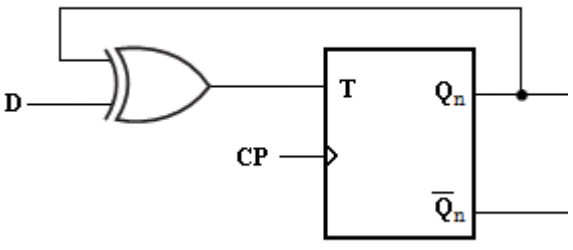
Input	Present state	Next state	Flip-Flop Input
D	**Q_n**	**Q_{n+1}**	**T**
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

K-map simplification



$$T = D\overline{Q}_n + \overline{D}Q_n$$
$$= D \oplus Q_n$$

Logic diagram



T to D Flip-Flop

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.

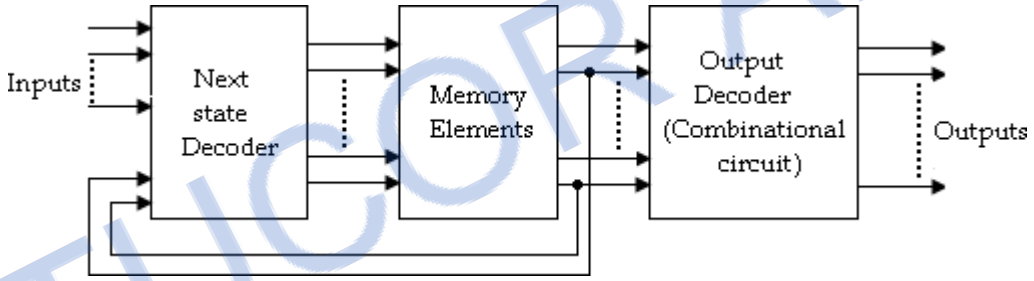
The synchronous or clocked sequential networks are represented by two models.

Moore model: The output depends only on the present state of the Flip-Flops.

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

Moore model:

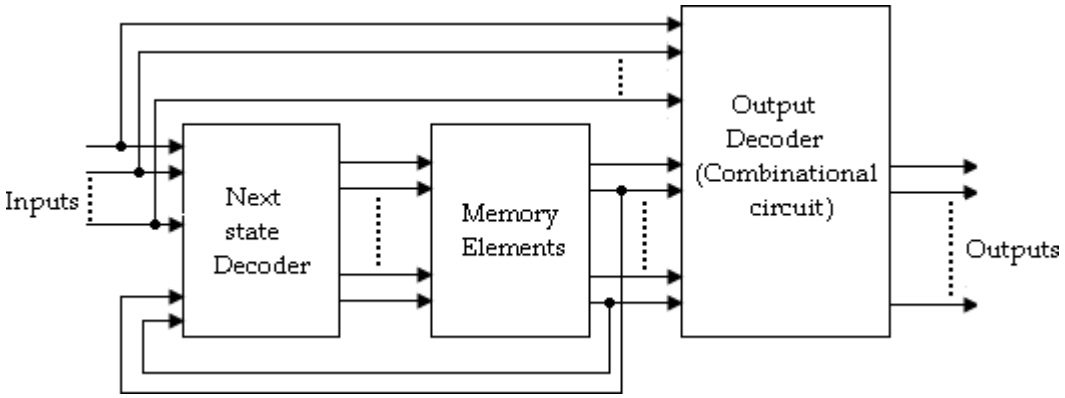
In the Moore model, the outputs are a function of the present state of the Flip-Flops only. The output depends only on present state of Flip-Flops, it appears only after the clock pulse is applied, i.e., it varies in synchronism with the clock input.



Moore model

Mealy model:

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



Mealy model

Difference between Moore and Mealy model

S.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	An input change 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.

ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUIT:

ANALYSIS PROCEDURE:

The synchronous sequential circuit analysis is summarizes 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.

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,

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

   $K_A = 1$ $K_B = 1$

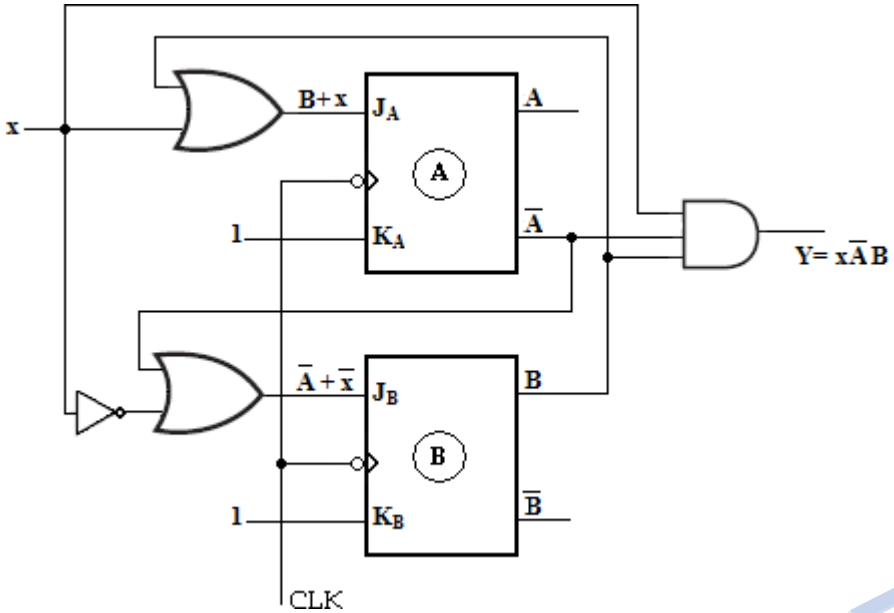
   and the circuit output function, $Y = xA'B$.

   a) Draw the logic diagram of the Mealy circuit,

   b) Tabulate the state table,

   c) Draw the state diagram.

Logic Diagram:



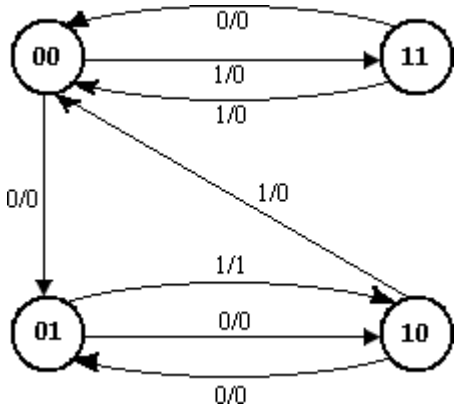
State table:

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

Reduced State Table:

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

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:

   $D_A = Ax + Bx$

   $D_B = A'x$

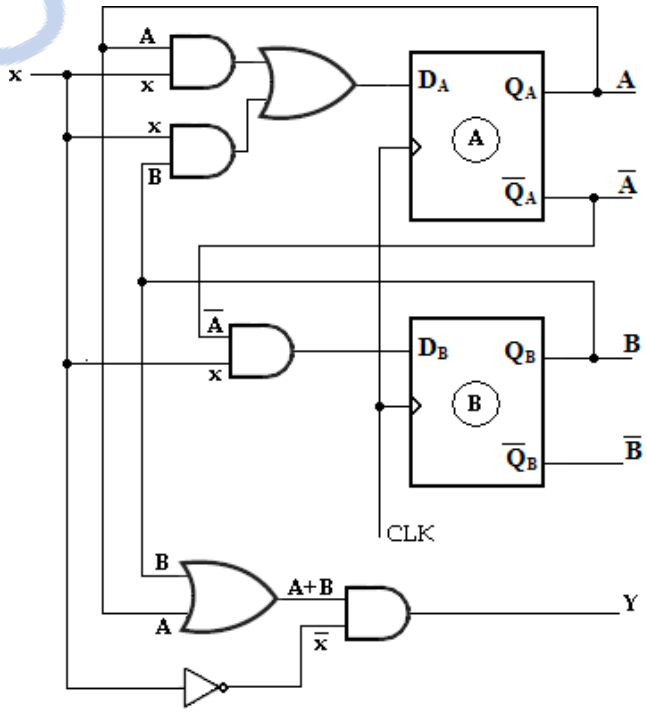
and the circuit output function is, $Y = (A + B)\,x'$.

(a) Draw the logic diagram of the circuit,

(b) Tabulate the state table,

(c) Draw the state diagram.

Soln:



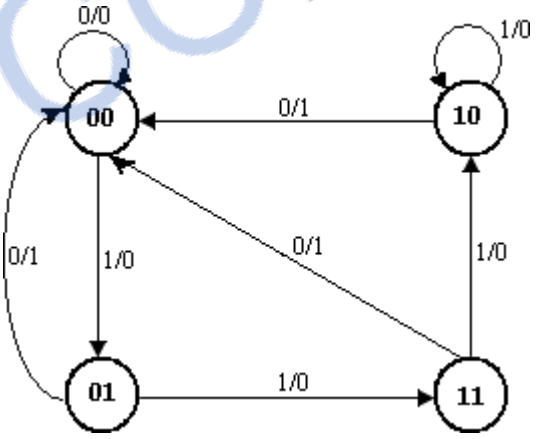
State Table:

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. A sequential circuit has two JK Flip-Flop A and B. the Flip-Flop input functions are: J_A= B J_B= x'

K_A= Bx' K_B= A⊕ x.

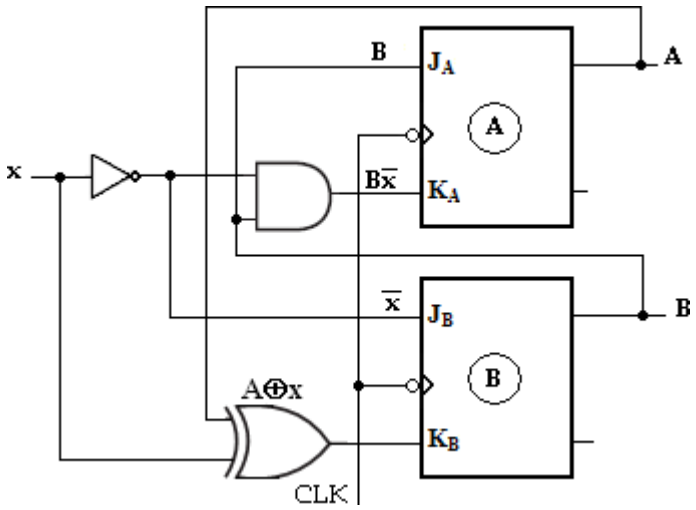
(a) Draw the logic diagram of the circuit,

(b) Tabulate the state table,

(c) Draw the state diagram.

Soln:

Logic diagram:



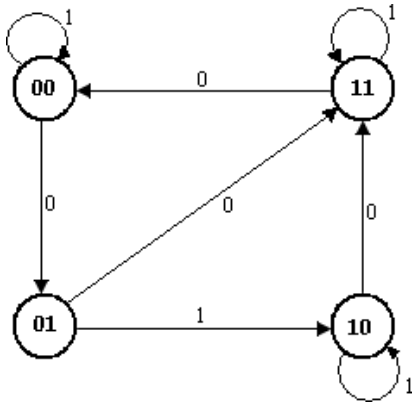
The output function is not given in the problem. The output of the Flip-Flops may be considered as the output of the circuit.

State table:

Present state		Input	Flip-Flop Inputs				Next state	
A	**B**	**x**	**J_A= B**	**K_A= Bx'**	**J_B= x'**	**K_B= A⊕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

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

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

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

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

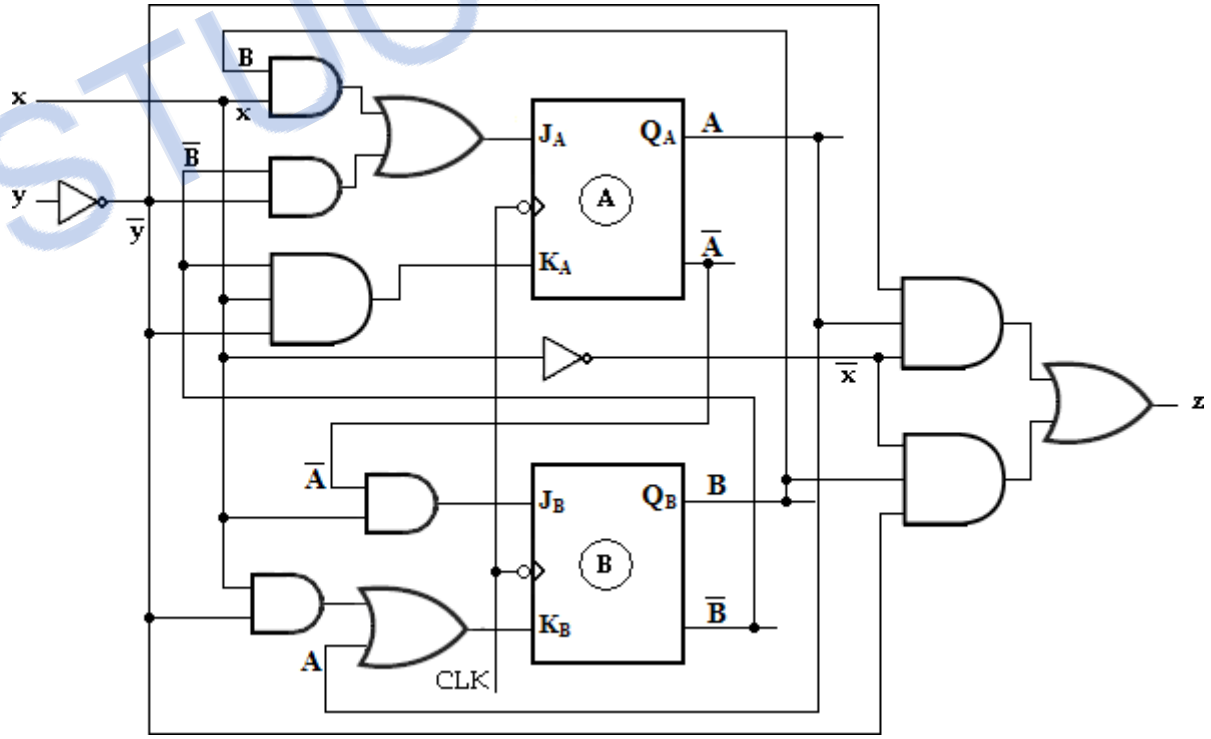
(a) Draw the logic diagram of the circuit

(b) Tabulate the state table.

(c) Derive the state equation.

Soln:

Logic 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

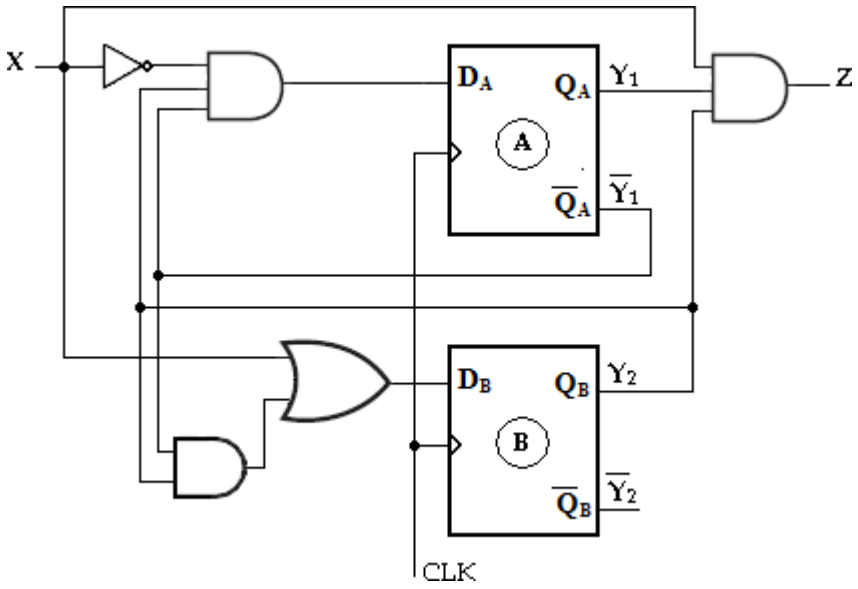
A (t+1) = Ax' + Ay + Bx + A'B'y'

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. 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,

$D_A = Y_1'Y_2X'$

$D_B = X + Y_1'Y_2$

The circuit output function is, $Z = Y_1Y_2X$.

State Table:

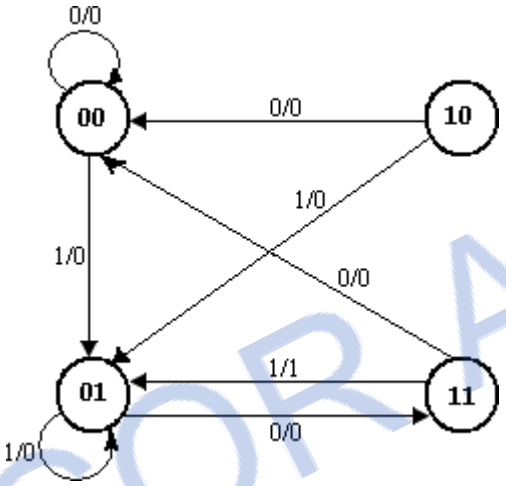
Present state		Input	Flip-Flop Inputs		Next state		Output
Y_1	Y_2	X	$D_A = Y_1'Y_2X'$	$D_B = X + Y_1'Y_2$	Y_1 (t+1)	Y_2 (t+1)	$Z = Y_1Y_2X$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Reduced State Table:

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

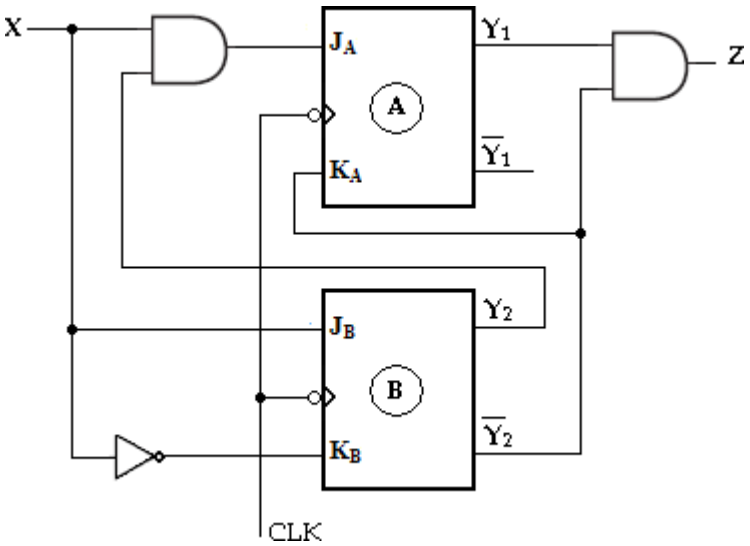
Second form of state table

State Diagram:



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:

$J_A = Y_2X$; $K_A = Y_2'$

$J_B = X$; $K_B = X'$ and output function, $Z = Y_1Y_2'$

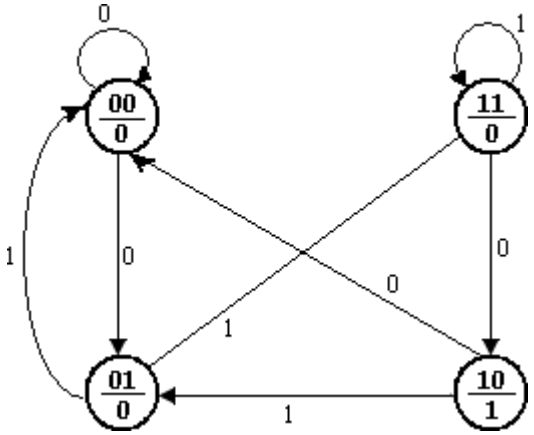
State table:

Present state		Input	Flip-Flop Inputs				Next state		Output
Y_1	Y_2	X	$J_A = Y_2X$	$K_A = Y_2'$	$J_B = X$	$K_B = X'$	Y_1 (t+1)	Y_2 (t+1)	$Z = Y_1Y_2'$
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
		X= 0		X= 1		Y
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

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:

   $T_A = Bx$ $T_B = x$

   $y = AB$

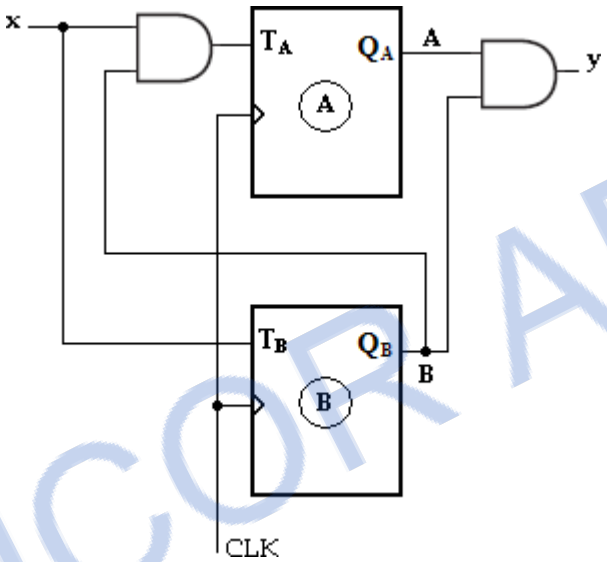
   (a) Draw the logic diagram of the circuit,

   (b) Tabulate the state table,

   (c) Draw the state diagram.

Soln:

Logic diagram:



State table:

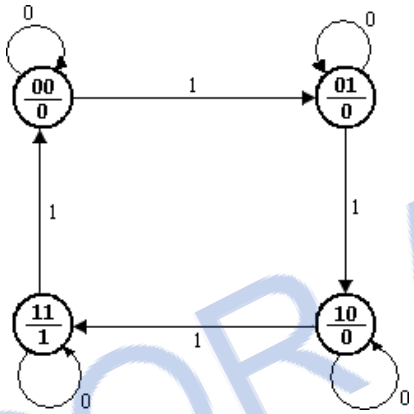
Present state		Input	Flip-Flop Inputs		Next state		Output
A	B	x	$T_A = Bx$	$T_B = x$	A (t+1)	B (t+1)	$y = AB$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1	1	1	0	0	1

Reduced State Table:

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:



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.

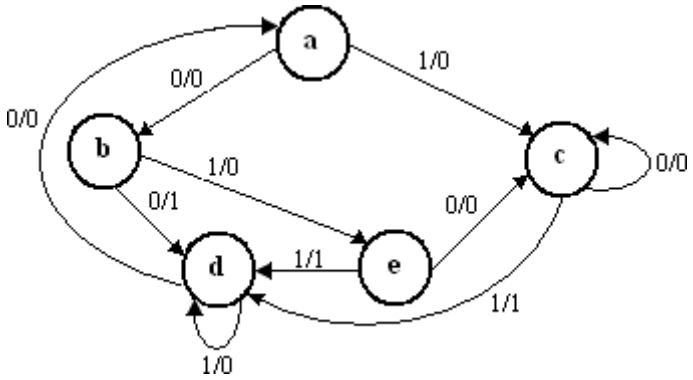
The two states are said to be redundant or equivalent, if every possible set of inputs generate exactly same output and same next state. When two states are equivalent, one of them can be removed without altering the input-output relationship.

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.

Examples:

1. Reduce the number of states in the following state diagram and draw the reduced state diagram.



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

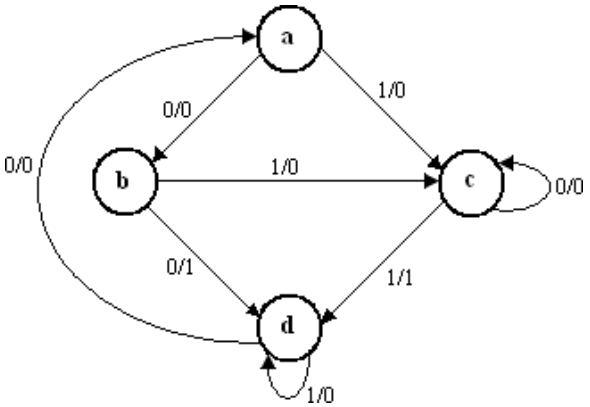
Step 2: Find equivalent states

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

Step 3: Draw state diagram



Reduced state diagram

2. 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

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.

3. 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.

4. 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**.

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**.

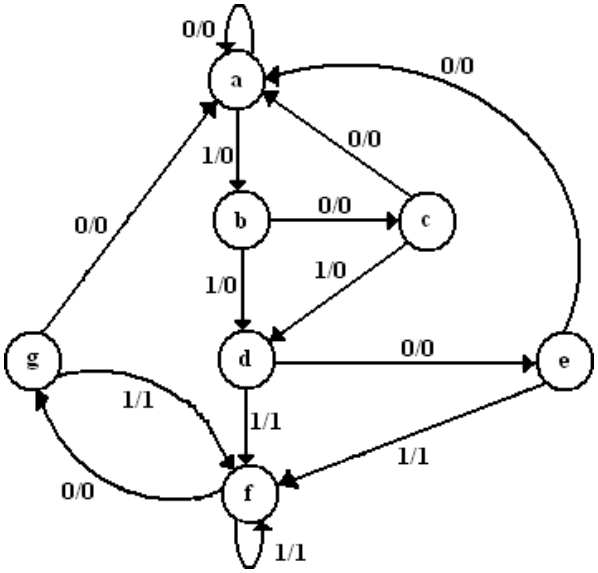
The final reduced state table-2 is shown below.

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.

5. 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.**

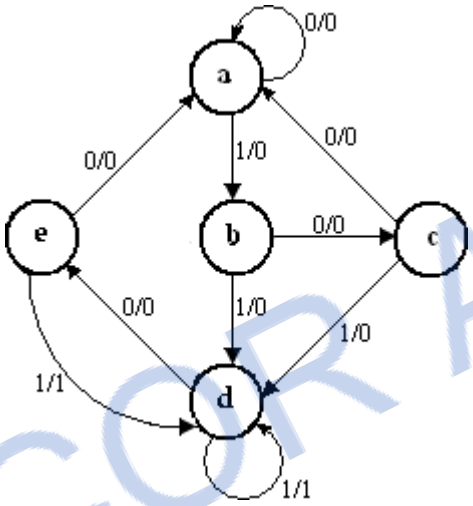
The final reduced state table-2 is shown below.

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

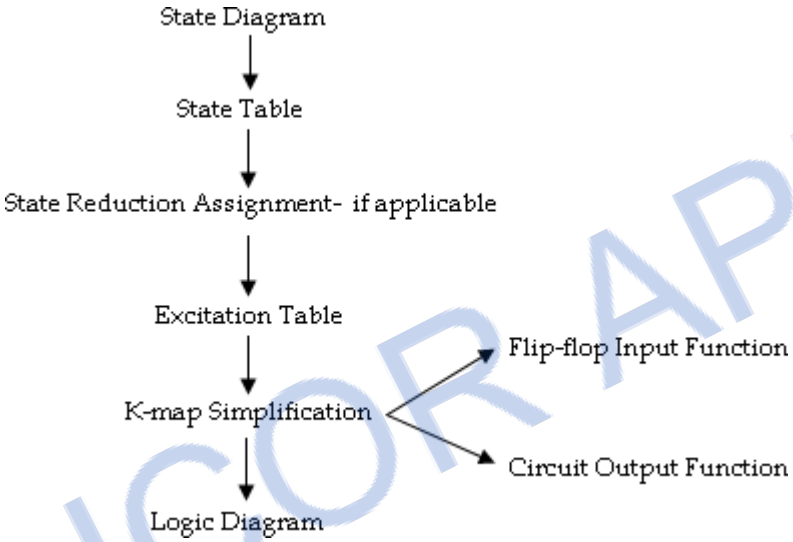
DESIGN OF SYNCHRONOUS SEQUENTIAL CIRCUITS:

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.

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 (Ex: Shift Registers)
T	Application involving complementation (Ex: Binary Counters)

3.10.2 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

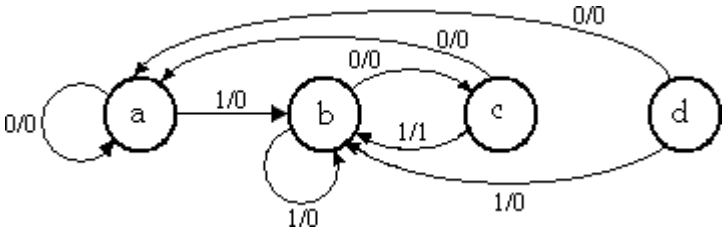
Excitation table for T Flip-Flop

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

Problems

1. 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

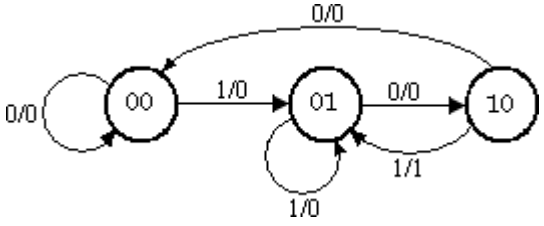
Reduced State Table:

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

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= 0; and c= 10.



Reduced State Diagram

Excitation Table:

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

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

For J_A

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

$J_A = X'B$

For K_A

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

$K_A = 1$

For Output

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

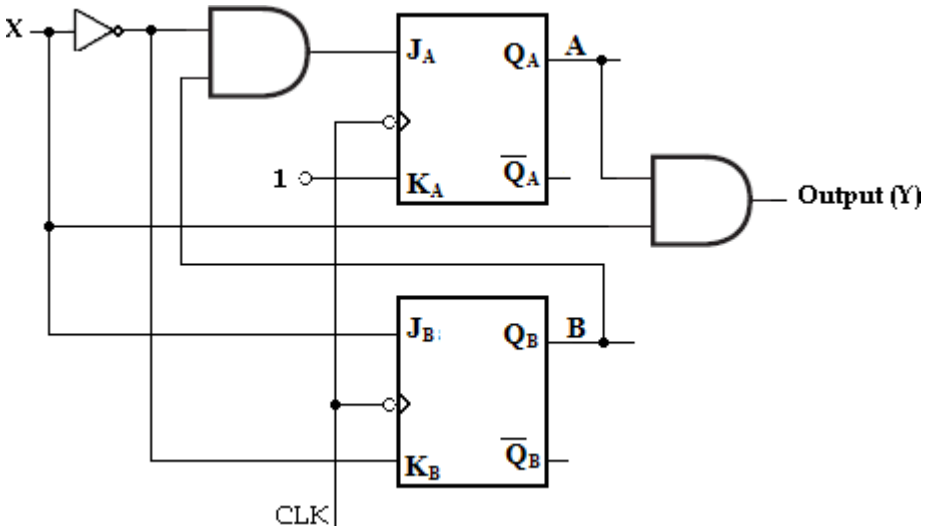
$J_B = X$

For K_B

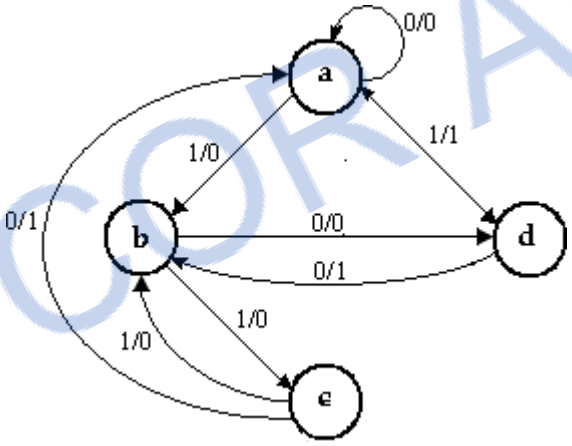
A \ BX	00	01	11	10
0	x	1	x	x
1	x	0	x	x

$K_B = X'$

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



2. 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:

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

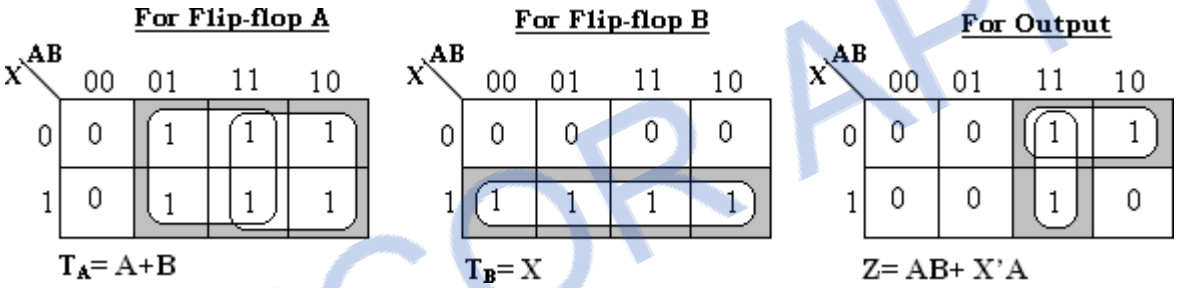
Even though a and c are having same next states for input X=0 and X=1, as the outputs are not same state reduction is not possible.

State Assignment:

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

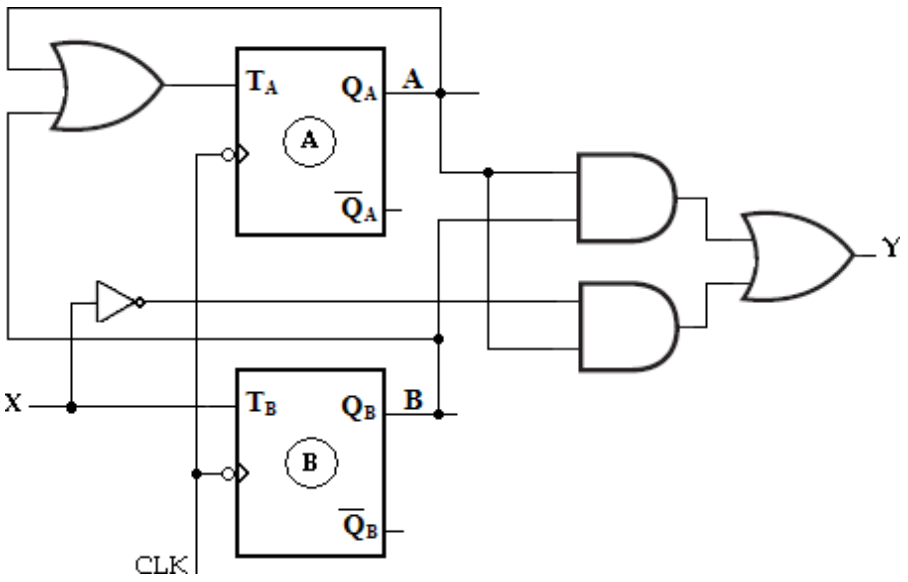
Input	Present state		Next state		Flip-Flop Inputs		Output
X	A	B	A	B	T_A	T_B	Y
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1
0	1	1	0	1	1	0	1
1	0	0	0	1	0	1	0
1	0	1	1	0	1	1	0
1	1	0	0	1	1	1	0
1	1	1	0	0	1	1	1

K-map simplification:



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

Logic Diagram:



SHIFT REGISTERS:

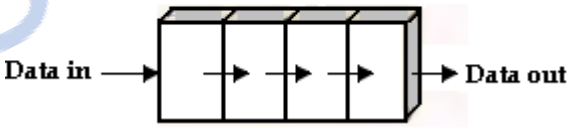
A register is simply a group of Flip-Flops that can be used to store a binary number. There must be one Flip-Flop for each bit in the binary number. For instance, a register used to store an 8-bit binary number must have 8 Flip-Flops.

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***.

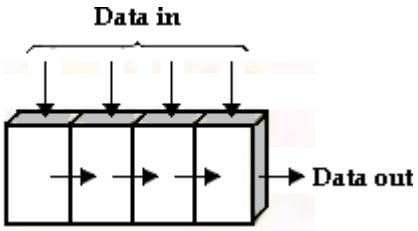
The bits in a binary number (data) can be removed from one place to another in either of two ways. The first method involves shifting the data one bit at a time in a serial fashion, beginning with either the most significant bit (MSB) or the least significant bit (LSB). This technique is referred to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

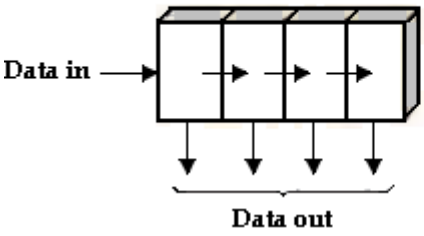
i. Serial in- serial out,
ii. Serial in- parallel out,
iii. Parallel in- serial out,
iv. Parallel in- parallel out.



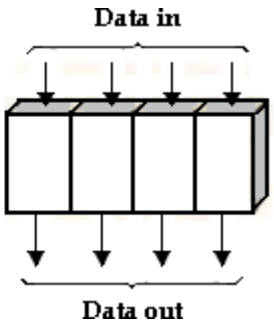
(i) Serial in- serial out



(iii) Parallel in- serial out



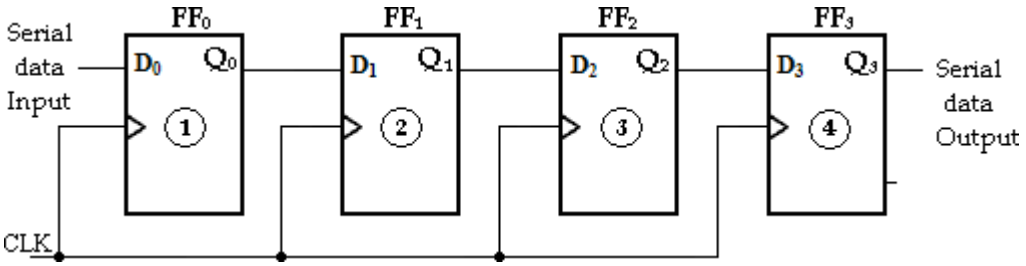
(iii) Serial in- parallel out



(iv) Parallel in- parallel out

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.



Serial-In Serial-Out Shift Register

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 FF_0. When the first clock pulse is applied, FF_0 is reset, thus storing the 0.

Next the second bit, which is a 1, is applied to the data input, making D=1 for FF_0 and D=0 for FF_1 because the D input of FF_1 is connected to the Q_0 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 FFl.

The third bit, a 0, is now put onto the data-input line, and a clock pulse is applied. The 0 is entered into FF_0, the 1 stored in FF_0 is shifted into FF_l, and the 0 stored in FF_1 is shifted into FF_2.

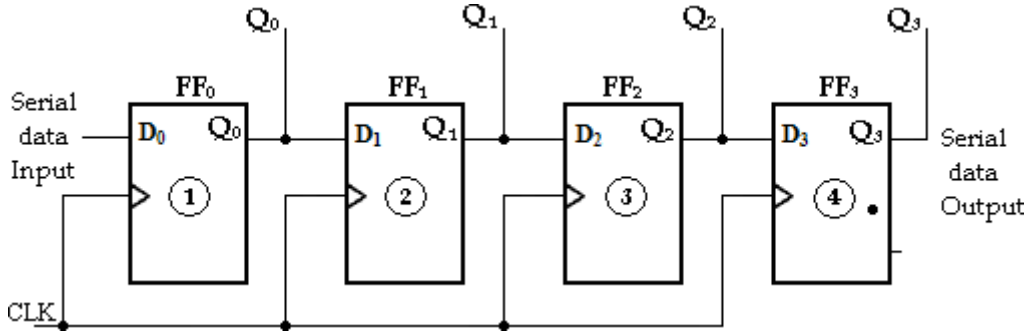
The last bit, a 1, is now applied to the data input, and a clock pulse is applied. This time the 1 is entered into FF0, the 0 stored in FF0 is shifted into FFl, the 1 stored in FF1 is shifted into FF2, and the 0 stored in FF2 is shifted into FF3. This completes the serial entry of the four bits into the shift register, where they can be stored for any length of time as long as the Flip-Flops have dc power.

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.

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.

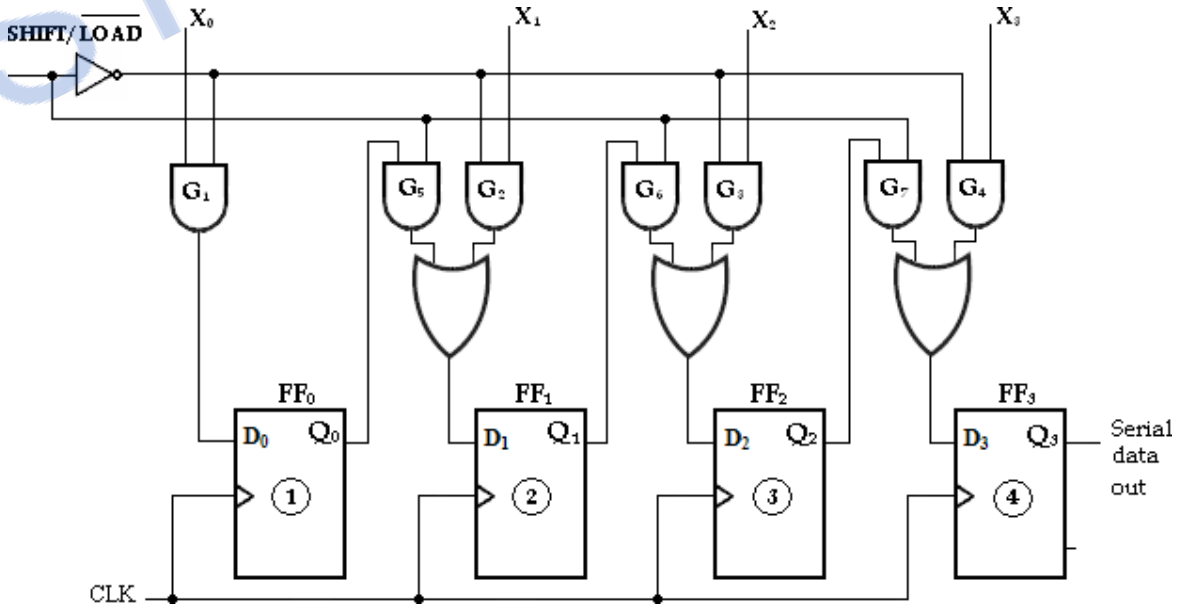


Serial-In parallel-Out Shift Register

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.



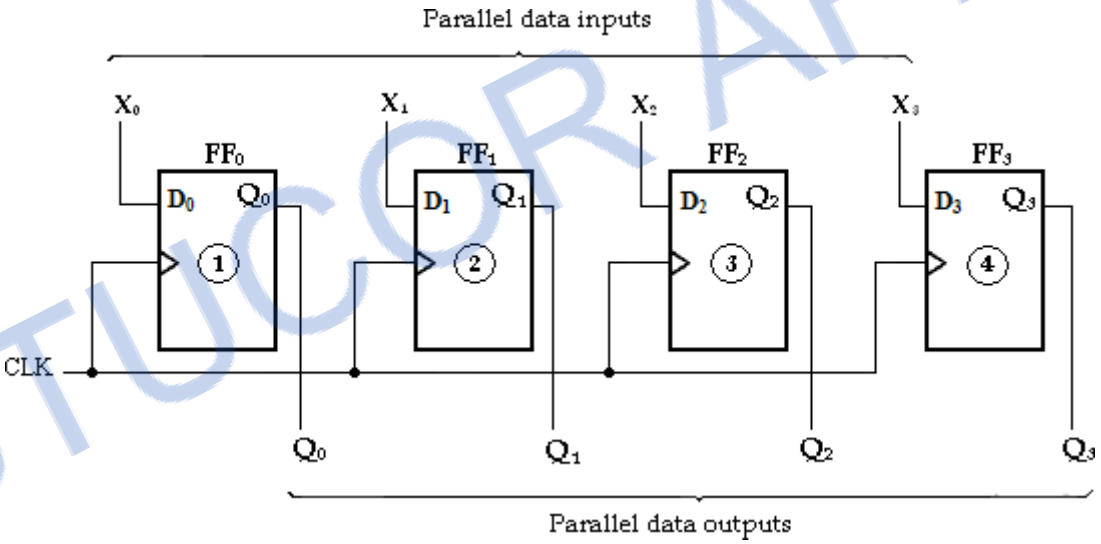
Parallel-In Serial-Out Shift Register

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.

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.



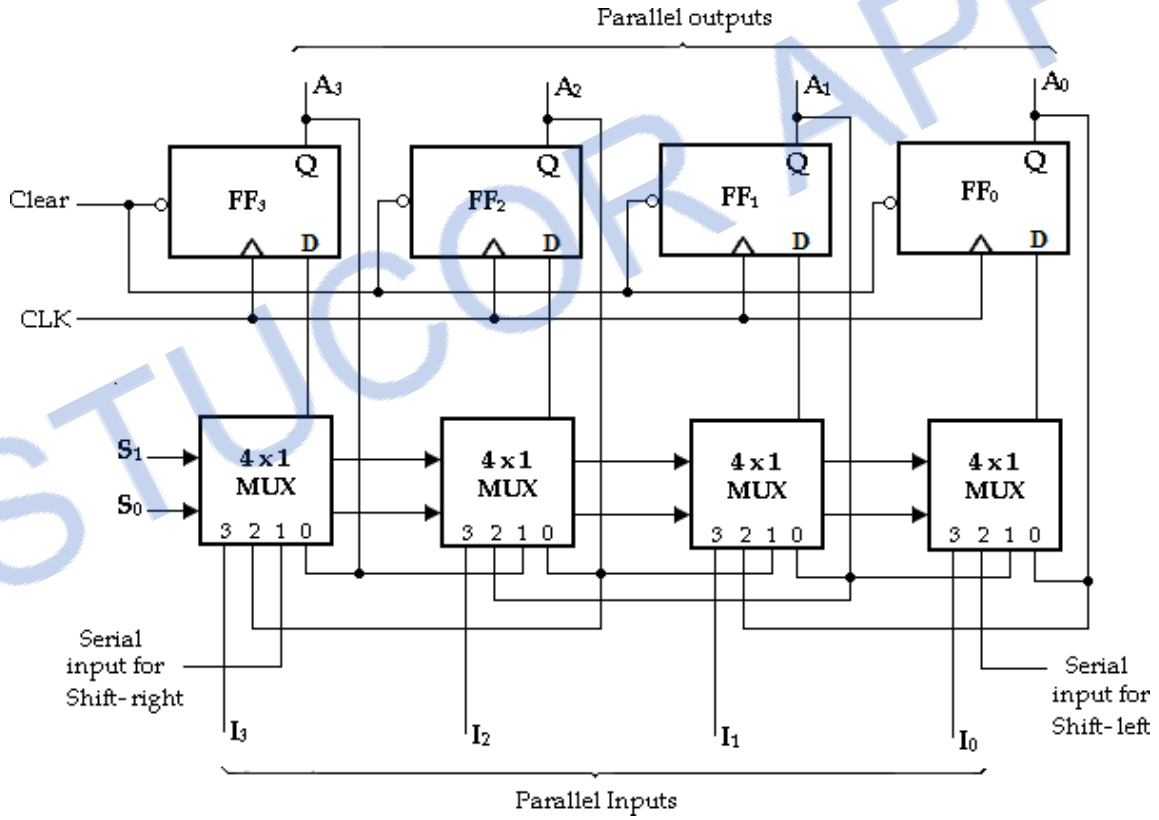
Parallel-In Parallel-Out Shift Register

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 re continuously applied.



4-Bit Universal Shift Register

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 S1 and S0 control the mode of the operation of the 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 lefter serial input transferred into Flip-Flop FF_3.

When S_1S_0= 10, a shift-left operation results with the right serial input going into Flip-Flop FF_1.

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

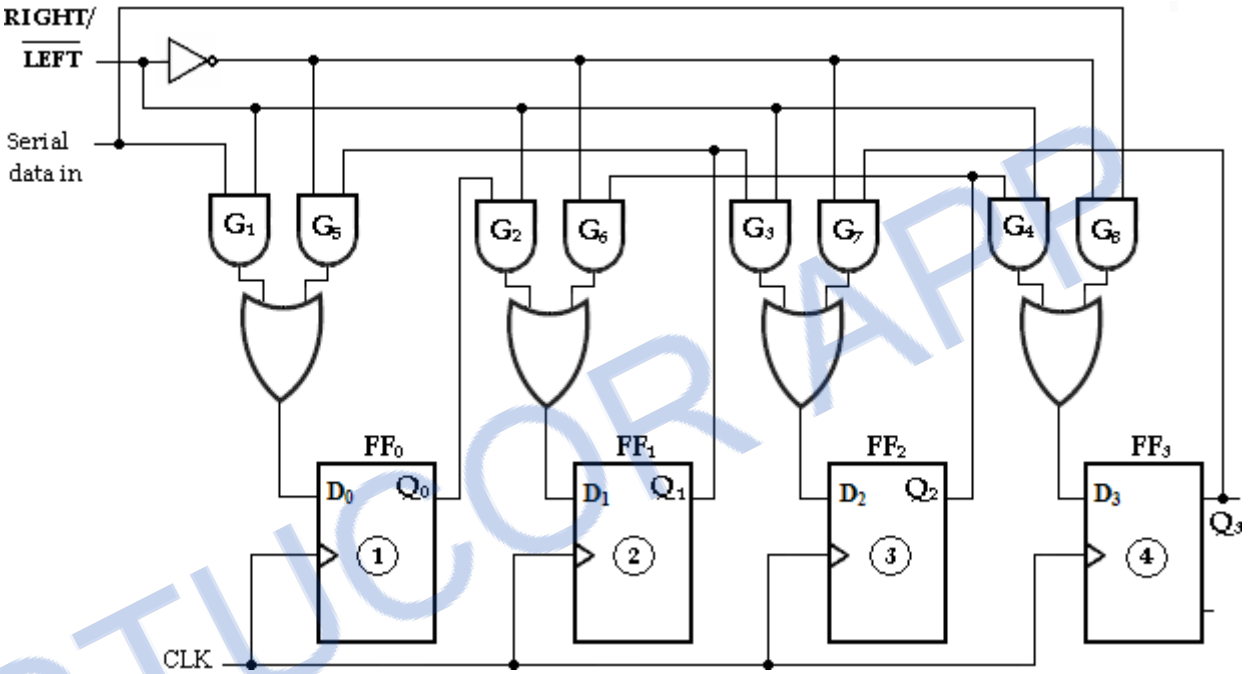
BI-DIRECTION SHIFT REGISTERS:

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 G_1, G_2, G_3 and G_4 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 G_5, G_6, G_7 and G_8 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

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.

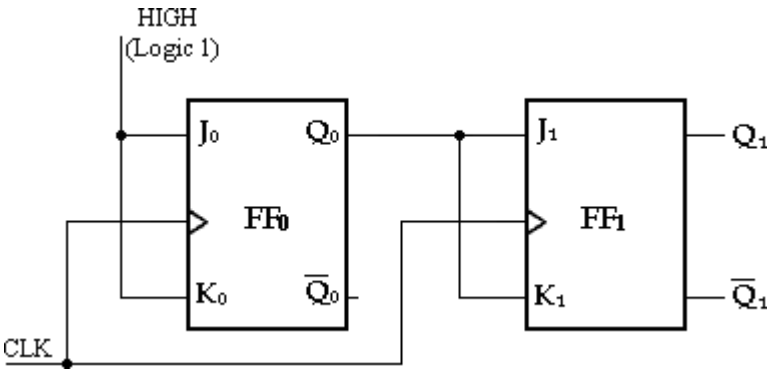
In synchronous counters, the clock input is connected to all of the Flip-Flops so that they are clocked simultaneously. Within each of these two categories, counters are classified primarily by the type of sequence, the number of states, or the number of Flip-Flops in the counter.

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 even for more number of states.	Design involves complex logic circuit 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.

2-Bit Synchronous Binary Counter

In this counter the clock signal is connected in parallel to clock inputs of both the Flip-Flops (FF_0 and FF_1). The output of FF_0 is connected to J_1 and K_1 inputs of the second Flip-Flop (FF_1).



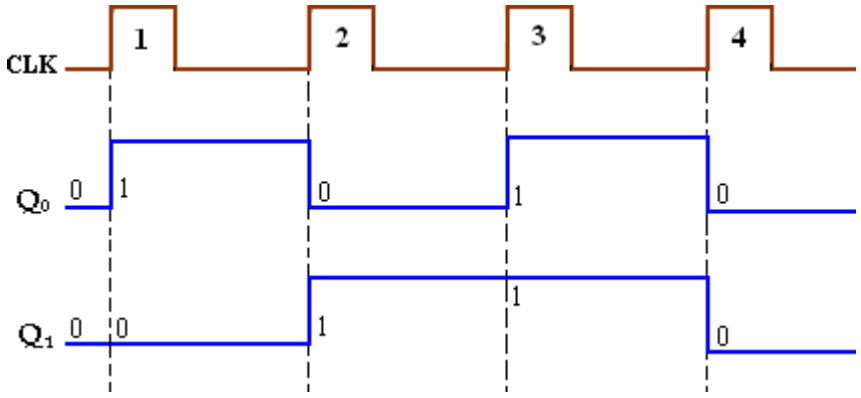
2-Bit Synchronous Binary Counter

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, FF_0 will toggle because $J_0 = k_0 = 1$, whereas FF_1 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, FF_0 will toggle and Q_0 will go LOW. Since FF_1 has a HIGH ($Q_0 = 1$) on its J_1 and K_1 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, FF_0 again toggles to the SET state ($Q_0 = 1$), and FF_1 remains SET ($Q_1 = 1$) because its J_1 and K_1 inputs are both LOW ($Q_0 = 0$). After this triggering edge, $Q_0 = 1$ and $Q_1 = 1$.

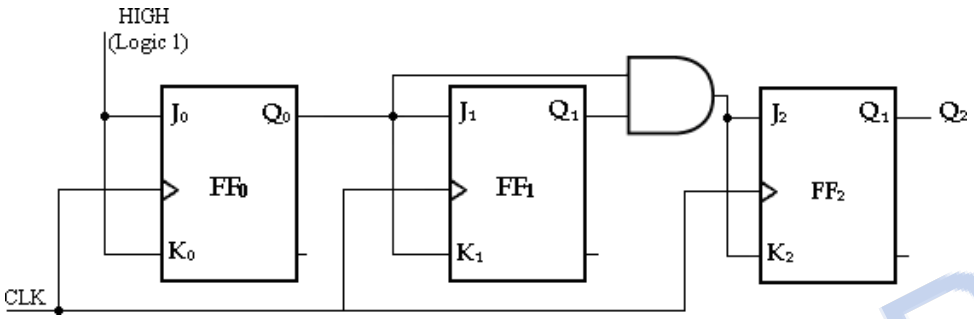
Finally, at the leading edge of CLK4, Q_0 and Q_1 go LOW because they both have a toggle condition on their J_1 and K_1 inputs. The counter has now recycled to its original state, $Q_0 = Q_1 = 0$.



Timing diagram

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 FF_0 (Q_0) 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, FF_0 must be held in the toggle mode by constant HIGH, on its J_0 and K_0 inputs.

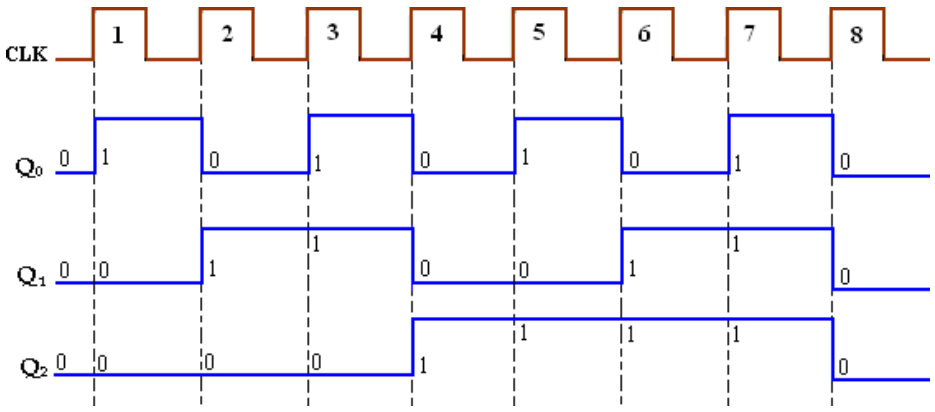


3- **Bit Synchronous Binary Counter**

The output of FF_1 (Q_1) goes to the opposite state following each time Q_0= 1. This change occurs at CLK2, CLK4, CLK6, and CLK8. The CLK8 pulse causes the counter to recycle. To produce this operation, Q_0 is connected to the J_1 and K_1 inputs of FF_1. When Q_0= 1 and a clock pulse occurs, FF_1 is in the toggle mode and therefore changes state. When Q_0= 0, FF_1 is in the no-change mode and remains in its present state.

The output of FF_2 (Q_2) changes state both times; it is preceded by the unique condition in which both Q_0 and Q_1 are HIGH. This condition is detected by the AND gate and applied to the J_2 and K_2 inputs of FF_2. Whenever both outputs Q_0= Q_1= 1, the output of the AND gate makes the J_2= K_2= 1 and FF_2 toggles on the following clock pulse. Otherwise, the J_2 and K_2 inputs of FF2 are held LOW by the AND gate output, FF_2 does not change state.

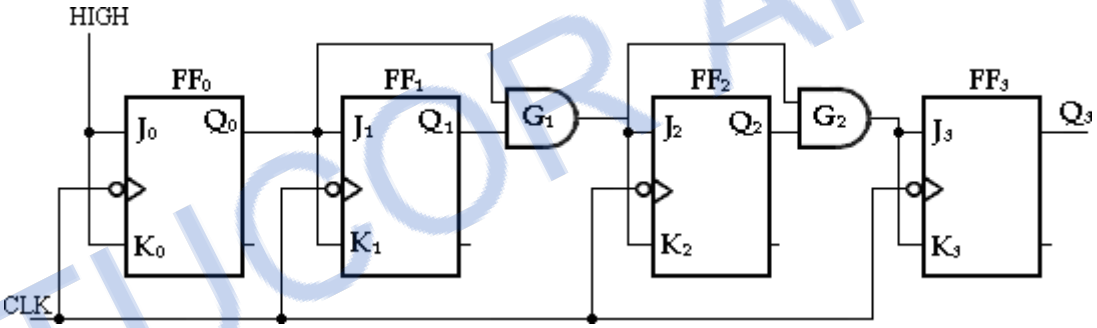
CLOCK Pulse	Q_2	Q_1	Q_0
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0



Timing diagram

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 G_2.

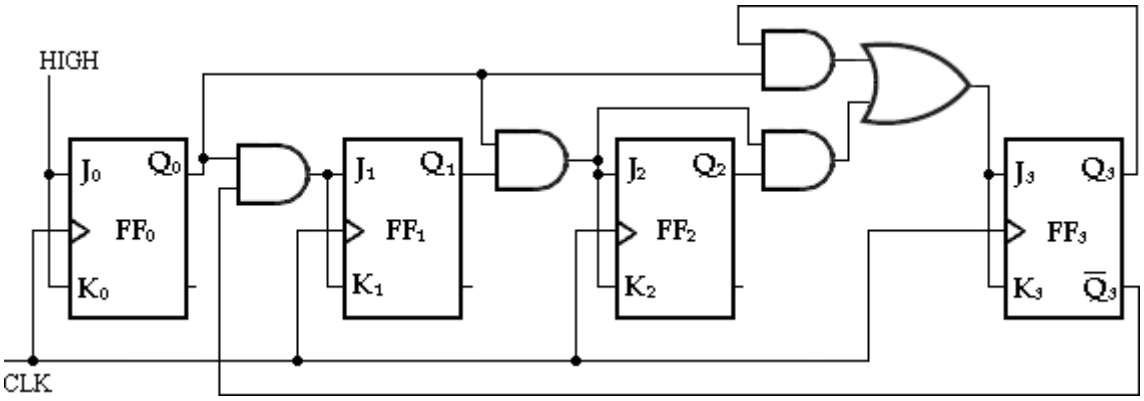


4- **Bit Synchronous Binary Counter**

Therefore, when $Q_0 = Q_1 = Q_2 = 1$, Flip-Flop FF_3 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.

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

- First, notice that FF_0 (Q_0) toggles on each clock pulse, so the logic equation for its J_0 and K_0 inputs is

$$J_0 = K_0 = 1$$

This equation is implemented by connecting J_0 and K_0 to a constant HIGH level.

- Next, notice from table, that FF_1 (Q_1) changes on the next clock pulse each time Q0 = 1 and Q3 = 0, so the logic equation for the J_1 and K_1 inputs is

$$J_1 = K_1 = Q_0Q_3'$$

- Flip-Flop 2 (Q_2) changes on the next clock pulse each time both $Q_0 = Q_1 = 1$. This requires an input logic equation as follows:

$$J_2 = K_2 = Q_0Q_1$$

This equation is implemented by ANDing Q_0 and Q_1 and connecting the gate output to the J_2 and K_2 inputs of FF_2.

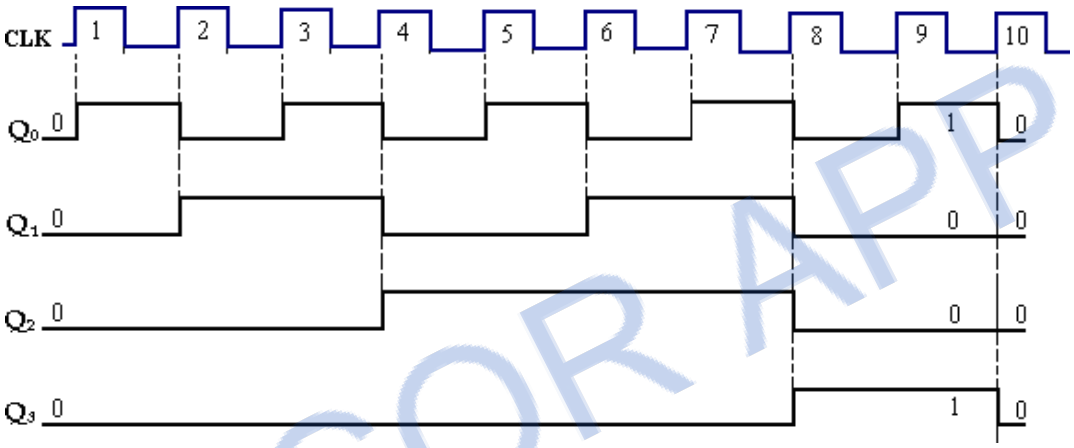
- Finally, FF_3 (Q_3) changes to the opposite state on the next clock pulse each time $Q_0 = 1$, $Q_1 = 1$, and $Q_2 = 1$ (state 7), or when $Q_0 = 1$ and $Q_1 = 1$ (state 9). The equation for this is as follows:

$$J_3 = K_3 = Q_0Q_1Q_2 + Q_0Q_3$$

This function is implemented with the AND/OR logic connected to the J_3 and K_3 inputs of FF_3.

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

The timing diagram for the decade counter is shown below.



Timing diagram

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 Q_0 for both the up and down sequences shows that FF_0 toggles on each clock pulse. Thus, the J_0 and K_0 inputs of FF_0 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	

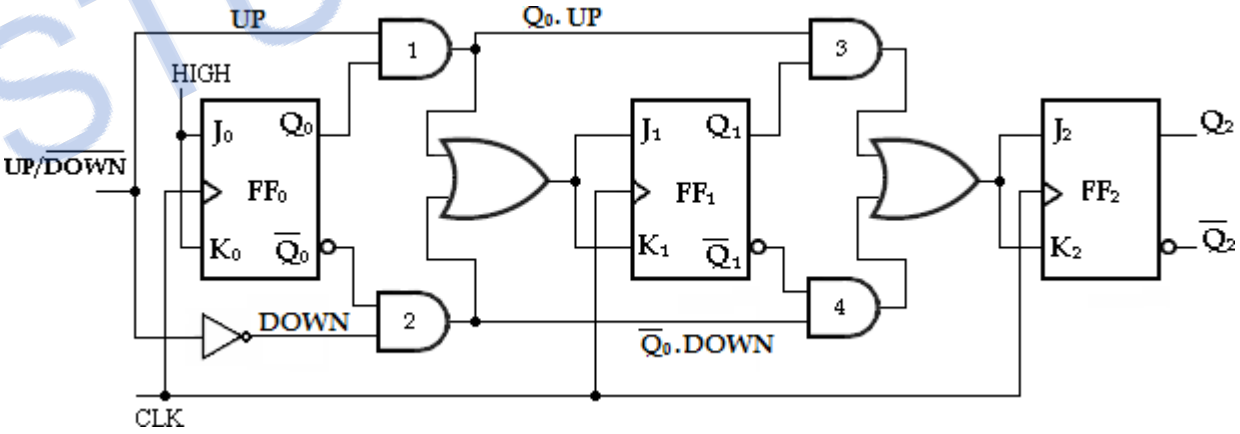
To form a synchronous UP/DOWN counter, the control input (UP/DOWN) is used to allow either the normal output or the inverted output of one Flip-Flop to the J and K inputs of the next Flip-Flop. When UP/DOWN= 1, the MOD 8 counter will count from 000 to 111 and UP/DOWN= 0, it will count from 111 to 000.

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.

$J_1 = K_1 = (Q_0.UP) + (Q_0'.DOWN)$

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



3-bit UP/DOWN Synchronous Counter

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 skin those are normally part of counting sequence.

n= 3

N= 8

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

(ii) MOD 5 Counter:

$2^n = N$

$2^n = 5$

$2^2 = 4$ less than N.

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

Therefore, 3 Flip-Flops are required.

(iii) MOD 10 Counter:

$2^n = N = 10$

$2^3 = 8$ less than N.

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

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

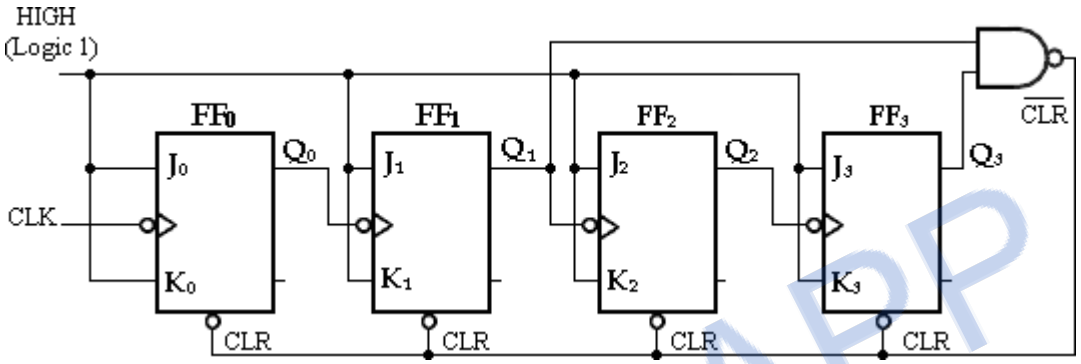
1. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

   $2^n \geq N$.
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

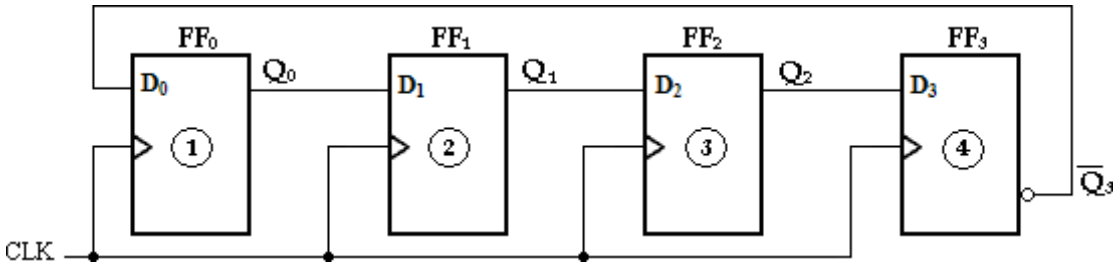
2.1 SHIFT REGISTER COUNTERS:

A shift register counter is basically a shift register with the serial output connected back to the serial input to produce special sequences. Two of the most common types of shift register counters are:

- Johnson counter (Shift Counter),
- Ring counter,

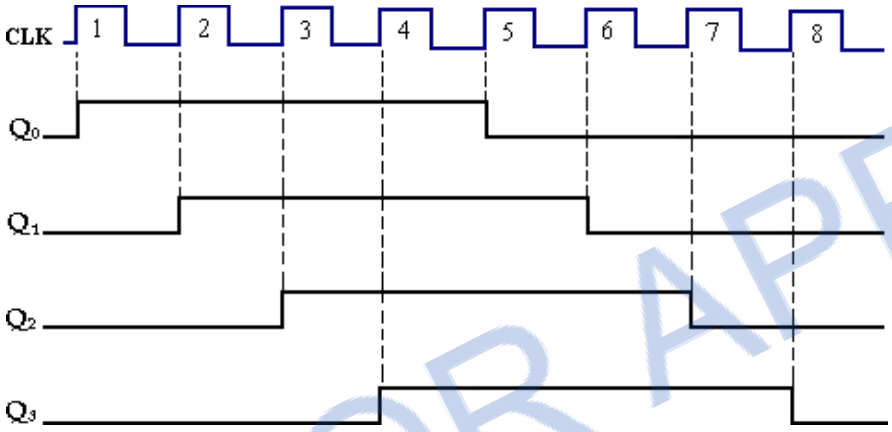
3.16.1 Johnson counter (Shift Counter):

In a Johnson counter the complement of the output of the last Flip-Flop is connected back to the D input of the first Flip-Flop. This feedback arrangement produces a characteristic sequence of states as shown in table below. The 4-bit sequence has a total of eight states, and that the 5-bit sequence has a total of ten states. In general, a Johnson counter will produce a modulus of **2n**, where 'n' is the number of stages in the counter.



4-Bit Johnson Counter

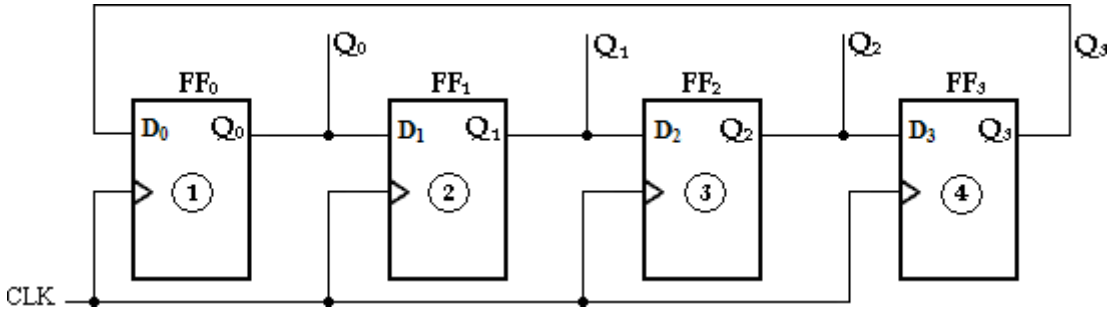
The Q output of each stage is connected to the D input of the next stage (assuming that D Flip-Flops are used). The complement output of the last stage is connected back to the D input of the first stage.



Time sequence for a 4-bit Johnson counter

3.16.2 Ring Counters:

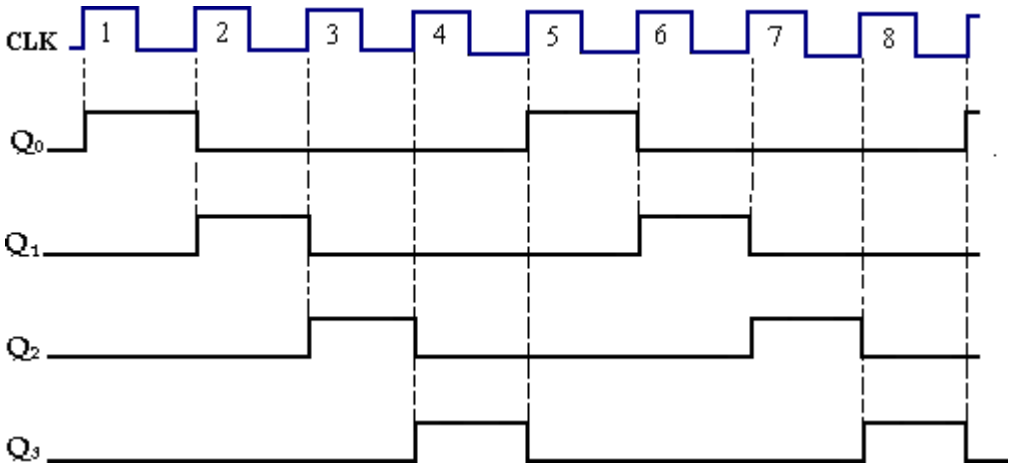
The ring counter utilizes one Flip-Flop for each state in its sequence. It has the advantage that decoding gates are not required. In the case of a l0-bit ring counter, there is a unique output for each decimal digit.



Ring counter

The output Q_0 sets D_1 input, Q_1 sets D_2, Q_2 sets D_3 and Q_3 is fed back to D_0. Because of these conditions, bits are shifted left one position per positive clock edge

and fed back to the input. All the Flip-Flops are clocked together. When CLR goes low then back to high, the output is 0000.



Time sequence for a Ring counter

The first positive clock edge shifts MSB to LSB position and other bits to one position left so that the output becomes Q= 0010. This process continues on second and third clock edge so that successive outputs are 0100 and 1000. The fourth positive clock edge starts the cycle all over again and the output is 0001. Thus the stored 1 bit follows a circular path (i.e., the stored 1 bits move left through all Flip-Flops and the final Flip-Flop sends it back to the first Flip-Flop). This action has given the name of ring counter.

2.2 DESIGN OF COUNTERS:

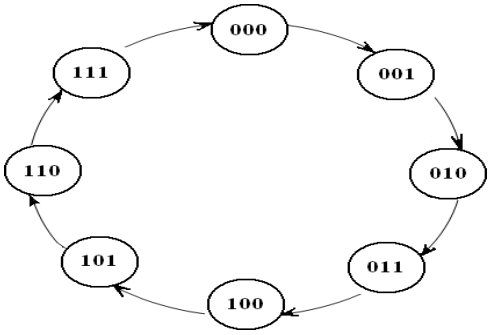
The procedure for design of counters as follows:

1. Specify the counter sequence and draw a state diagram.
2. Derive a next-state table from the state diagram.
3. Make the state assignment and develop a transition table showing the flip-flop inputs required.
4. Draw the K-maps for each input of each Flip-Flop.
5. Derive the logic expression for each Flip-Flop input from the K-maps.
6. Implement the expressions with combinational logic and combine with the Flip-Flops to form the counter.

Examples:

1. Using JK Flip-Flops, design a synchronous counter which counts in the sequence, **000, 001, 010, 011, 100, 101, 110, 111, 000.**

Step 1: State Diagram



Step 2: Excitation Table :

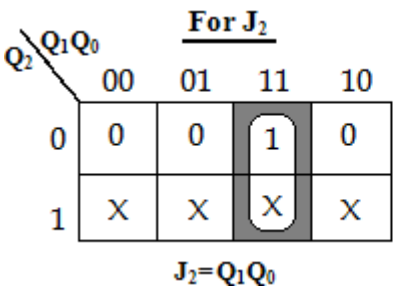
Excitation Table for JK Flip-Flop:

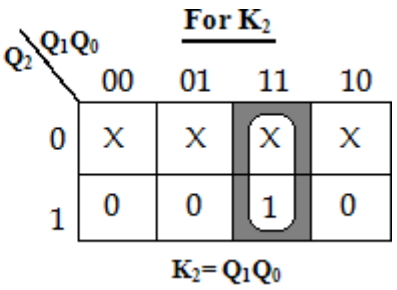
Present State	Next State	Inputs	
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

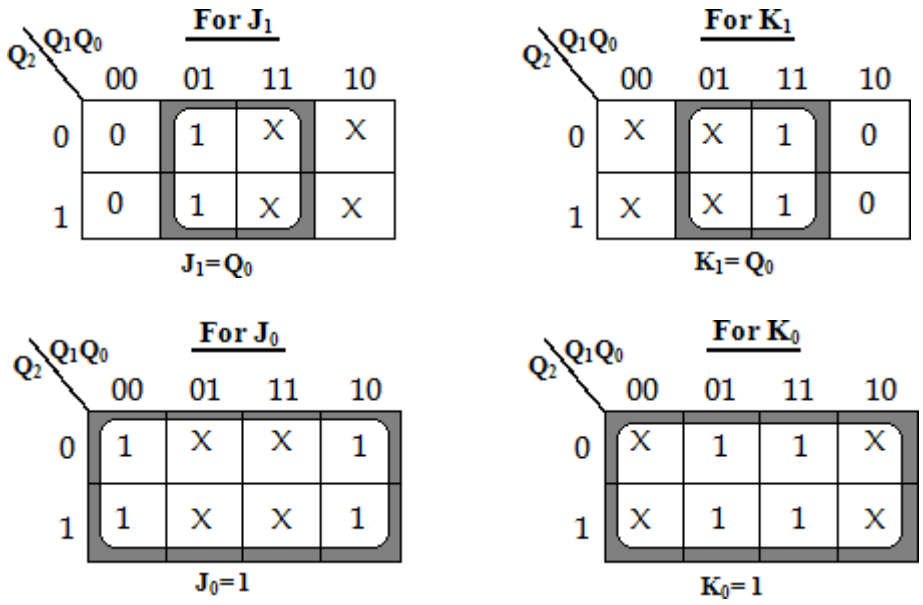
Excitation Table for Counter:

Present State			Next State			Flip-Flop Inputs					
Q_2	Q_1	Q_0	q_2	q_1	q_0	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	1	0	x	0	x	1	x
0	0	1	0	1	0	0	x	1	x	x	1
0	1	0	0	1	1	0	x	x	0	1	x
0	1	1	1	0	0	1	x	x	1	x	1
1	0	0	1	0	1	x	0	0	x	1	x
1	0	1	1	1	0	x	0	1	x	x	1
1	1	0	1	1	1	x	0	x	0	1	x
1	1	1	0	0	0	x	1	x	1	x	1

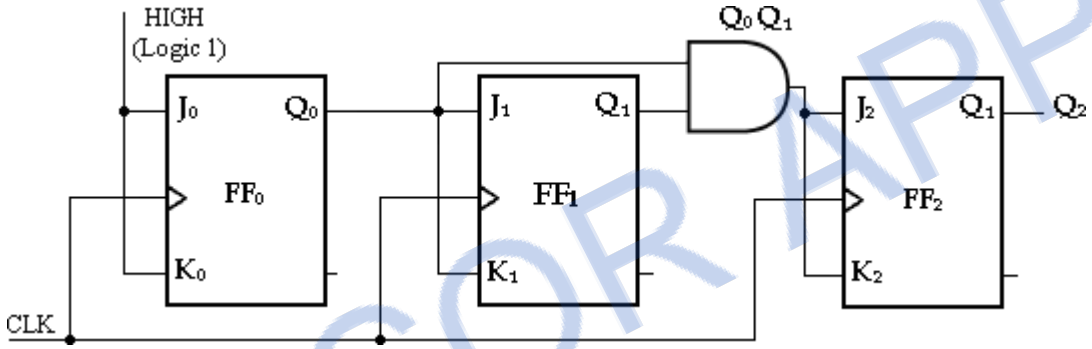
Step 3: K-map Simplification







<u>Step 4</u>: Logic Diagram



2. Design and explain the working of a synchronous MOD-3 counter.

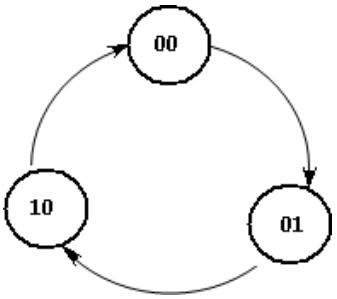
<u>Soln</u>:

$2^n \geq N = 3$

$2^2 > 3$.

Therefore, 2 Flip-Flops are required.

State Diagram:



Excitation Table:

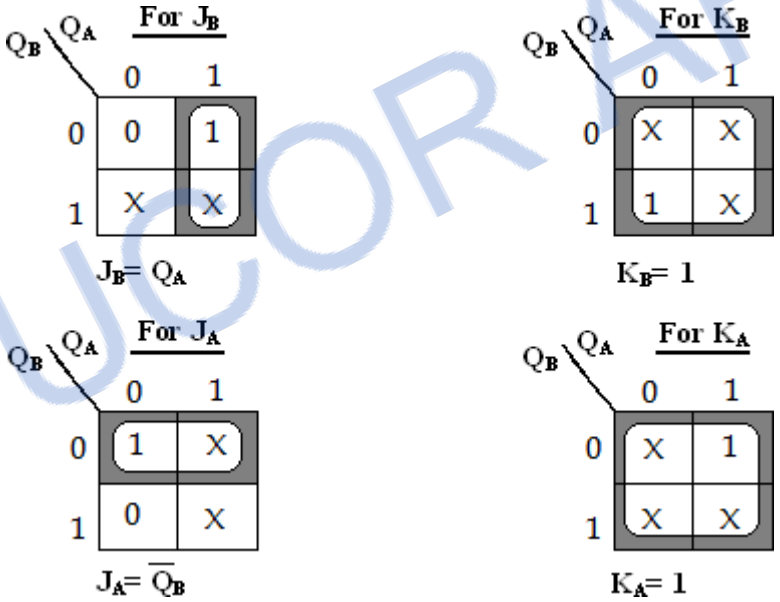
Excitation Table for JK Flip-Flop:

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

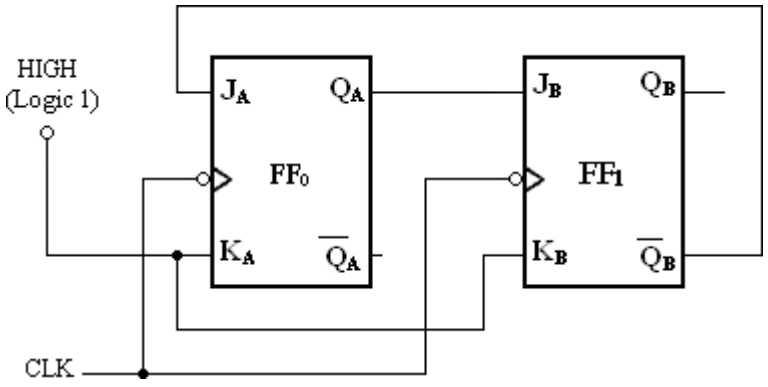
Excitation Table for Counter:

Present State		Next State		Flip-Flop Inputs			
Q_B	Q_A	Q_{B+1}	Q_{A+1}	J_B	K_B	J_A	K_A
0	0	0	1	0	x	1	x
0	1	1	0	1	x	x	1
1	0	0	0	x	1	0	x

K-map Simplification:



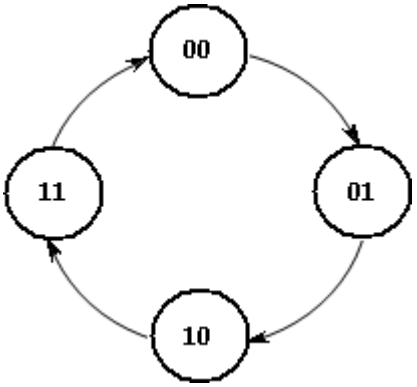
Logic Diagram:



3. Design a synchronous counter with states 0, 1, 2, 3, 0, 1, ...using JK Flip-Flops.

Soln:

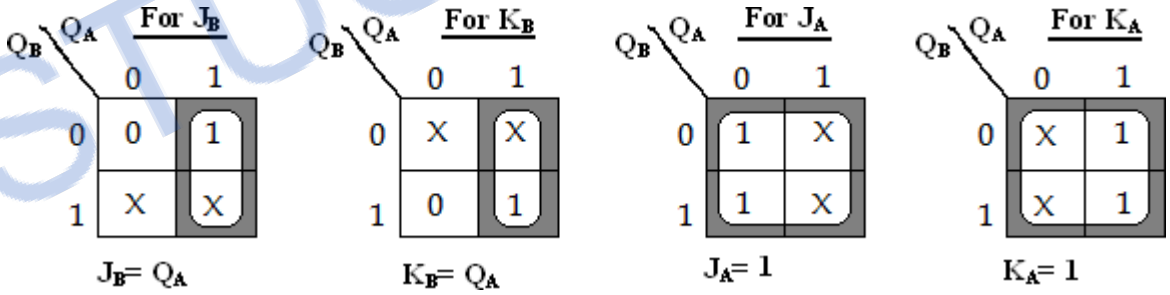
State Diagram:



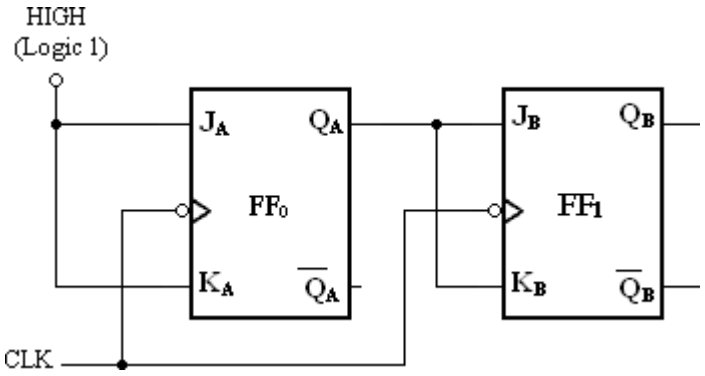
Excitation Table for Counter:

Present State		Next State		Flip-Flop Inputs			
Q_B	Q_A	Q_{B+1}	Q_{A+1}	J_B	K_B	J_A	K_A
0	0	0	1	0	x	1	x
0	1	1	0	1	x	x	1
1	0	1	1	x	0	1	x
1	1	0	0	x	1	x	1

K-map Simplification:



Logic Diagram:



4. Design a MOD-7 synchronous counter using JK Flip-Flops. Write excitation table and state table.

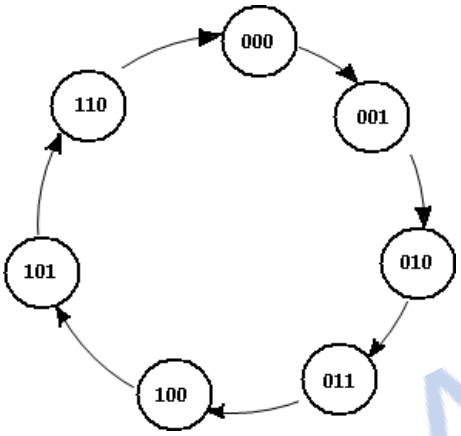
Soln:

$2^n \geq N = 7$

$2^3 > 8$.

Therefore, 3 Flip-Flops are required.

State Diagram:



Excitation Table:

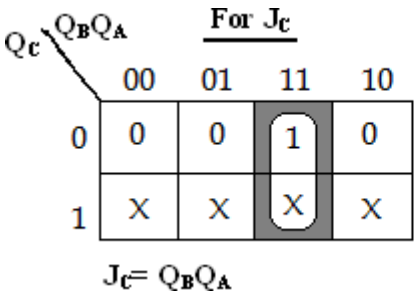
Excitation Table for JK Flip-Flop:

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

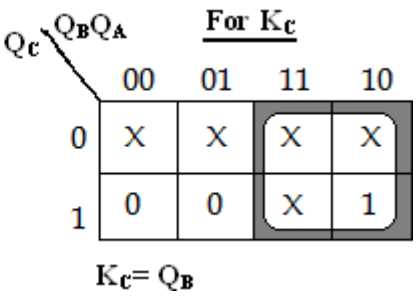
Excitation Table for Counter:

Present State			Next State			Flip-Flop Inputs					
Q_C	Q_B	Q_A	Q_{C+1}	Q_{B+1}	Q_{A+1}	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	1	0	x	0	x	1	x
0	0	1	0	1	0	0	x	1	x	x	1
0	1	0	0	1	1	0	x	x	0	1	x
0	1	1	1	0	0	1	x	x	1	x	1
1	0	0	1	0	1	x	0	0	x	1	x
1	0	1	1	1	0	x	0	1	x	x	1
1	1	0	0	0	0	x	1	x	1	0	x

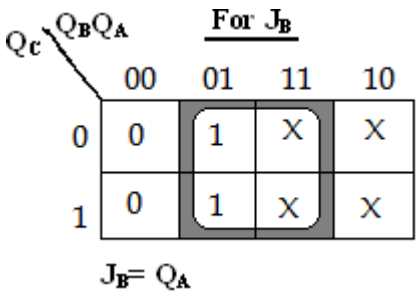
K-map Simplification:



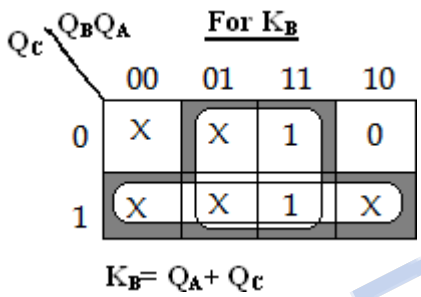
$J_C = Q_BQ_A$



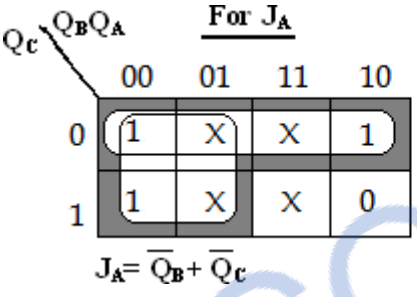
$K_C = Q_B$



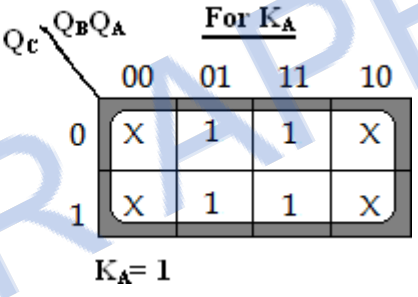
$J_B = Q_A$



$K_B = Q_A + Q_C$

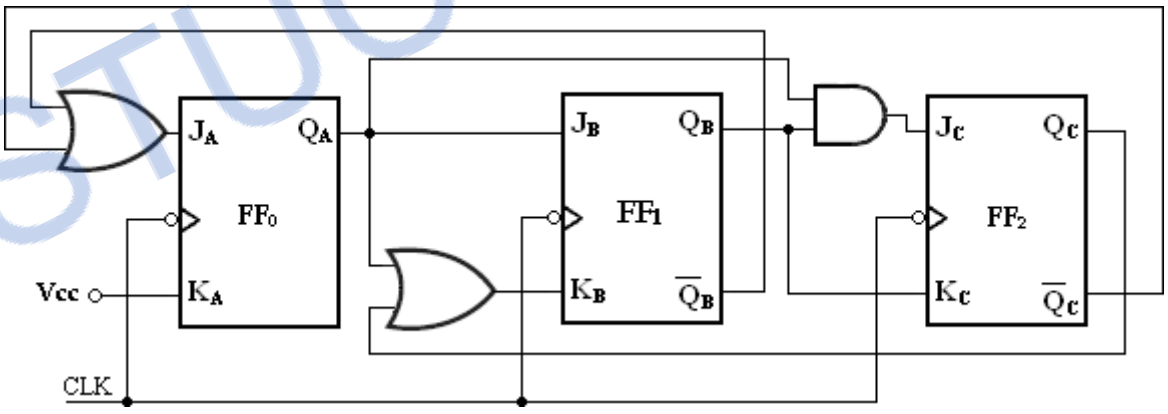


$J_A = \overline{Q}_B + \overline{Q}_C$



$K_A = 1$

Logic Diagram:



5. Design a MOD-10 synchronous counter using JK Flip-Flops. Write excitation table and state table.

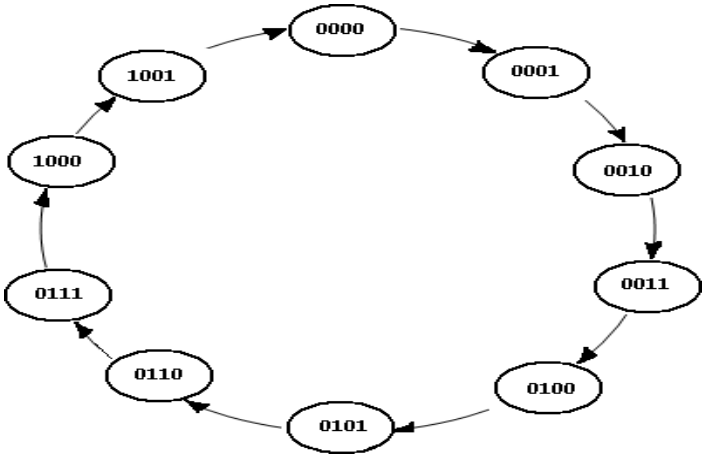
Soln:

$2^n \geq N = 10$

$2^4 > 10.$

Therefore, 4 Flip-Flops are required.

State Table:



Excitation Table:

Excitation Table for JK 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 Counter:

Present State				Next State				Flip-Flop Inputs							
Q_D	Q_C	Q_B	Q_A	Q_{D+1}	Q_{C+1}	Q_{B+1}	Q_{A+1}	J_D	K_D	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	0	0	1	0	x	0	x	0	x	1	x
0	0	0	1	0	0	1	0	0	x	0	x	1	x	x	1
0	0	1	0	0	0	1	1	0	x	0	x	x	0	1	x
0	0	1	1	0	1	0	0	0	x	1	x	x	1	x	1
0	1	0	0	0	1	0	1	0	x	x	0	0	x	1	x
0	1	0	1	0	1	1	0	0	x	x	0	1	x	x	1
0	1	1	0	0	1	1	1	0	x	x	0	x	0	1	x
0	1	1	1	1	0	0	0	1	x	x	1	x	1	x	1
1	0	0	0	1	0	0	1	x	0	0	x	0	x	1	x
1	0	0	1	0	0	0	0	x	1	0	x	0	x	x	1

K-map Simplification:

For J_D

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	X	X	X	X
10	X	X	X	X

$J_D = Q_CQ_BQ_A$

For K_D

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	X	X
10	0	1	X	X

$K_D = Q_A$

For J_C

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	0	0	1	0
01	X	X	X	X
11	X	X	X	X
10	0	0	X	X

$J_C = Q_BQ_A$

For K_C

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	X	X	X	X
01	0	0	1	0
11	X	X	X	X
10	X	X	X	X

$K_C = Q_BQ_A$

For J_B

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	0	1	X	X
01	0	1	X	X
11	X	X	X	X
10	0	0	X	X

$J_B = \overline{Q}_DQ_A$

For K_B

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	X	X	1	0
01	X	X	1	0
11	X	X	X	X
10	X	X	X	X

$K_B = Q_A$

For J_A

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	1	X	X	1
01	1	X	X	1
11	X	X	X	X
10	1	X	X	X

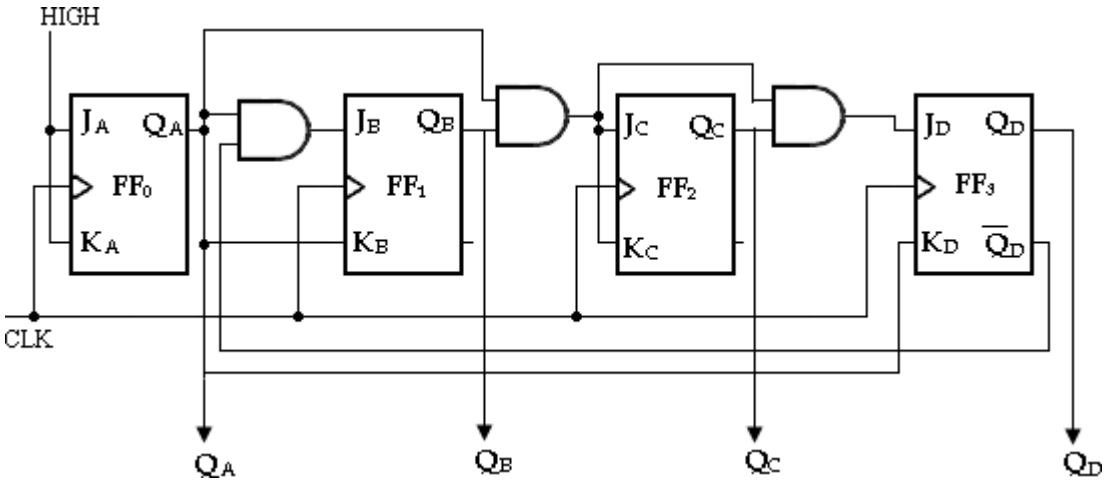
$J_A = 1$

For K_A

Q_DQ_C \ Q_BQ_A	00	01	11	10
00	X	1	1	X
01	X	1	1	X
11	X	X	X	X
10	X	X	X	X

$K_A = 1$

Logic Diagram:

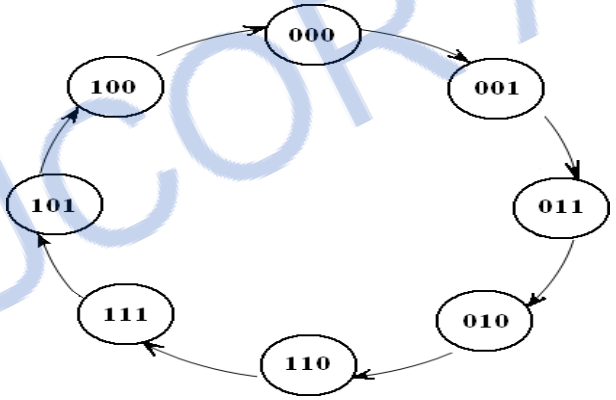


6. Design a synchronous 3-bit gray code up counter with the help of excitation table.

Soln:

Gray code sequence: 000, 001, 011, 010, 110, 111, 101, 100.

State Diagram:



Excitation Table:

Present State			Next State			Flip-Flop Inputs					
Q_C	Q_B	Q_A	Q_{C+1}	Q_{B+1}	Q_{A+1}	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	1	0	x	0	x	1	x
0	0	1	0	1	1	0	x	1	x	x	0
0	1	1	0	1	0	0	x	x	0	x	1
0	1	0	1	1	0	1	x	x	0	0	x
1	1	0	1	1	1	x	0	x	0	1	x
1	1	1	1	0	1	x	0	x	1	x	0
1	0	1	1	0	0	x	0	0	x	x	1
1	0	0	0	0	0	x	1	0	x	0	x

K-map Simplification:

For J_C

Q_C \ Q_BQ_A	00	01	11	10
0	0	0	0	1
1	X	X	X	X

$J_C = Q_B\overline{Q_A}$

For K_C

Q_C \ Q_BQ_A	00	01	11	10
0	X	X	X	X
1	1	0	0	0

$K_C = \overline{Q_B}\overline{Q_A}$

For J_B

Q_C \ Q_BQ_A	00	01	11	10
0	0	1	X	X
1	0	0	X	X

$J_B = \overline{Q_C}Q_A$

For K_B

Q_C \ Q_BQ_A	00	01	11	10
0	X	X	0	0
1	X	X	1	0

$K_B = Q_CQ_A$

For J_A

Q_C \ Q_BQ_A	00	01	11	10
0	1	X	X	0
1	0	X	X	1

$J_A = \overline{Q_B}\overline{Q_C} + Q_BQ_C$

$Q_B \odot Q_C$

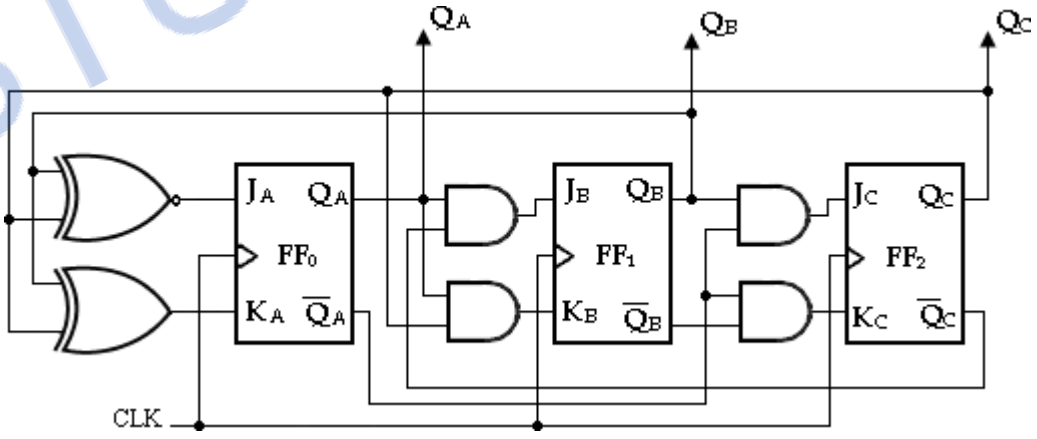
For K_A

Q_C \ Q_BQ_A	00	01	11	10
0	X	0	1	X
1	X	1	0	X

$K_A = Q_B\overline{Q_C} + \overline{Q_B}Q_C$

$Q_B \oplus Q_C$

Logic Diagram:



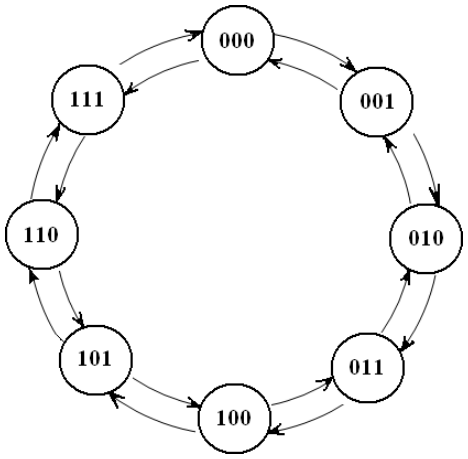
7. Design a 3 bit (MOD 8) Synchronous UP/DOWN counter.

Soln:

When UP/DOWN= 1, UP mode,

UP/DOWN= 0, DOWN mode.

State Diagram:



Excitation Table:

Input Up/Down	Present State			Next State			A		B		C	
	Q_A	Q_B	Q_C	Q_{A+1}	Q_{B+1}	Q_{C+1}	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	1	1	1	1	x	1	x	1	x
0	1	1	1	1	1	0	x	0	x	0	x	1
0	1	1	0	1	0	1	x	0	x	1	1	x
0	1	0	1	1	0	0	x	0	0	x	x	1
0	1	0	0	0	1	1	x	1	1	x	1	x
0	0	1	1	0	1	0	0	x	x	0	x	1
0	0	1	0	0	0	1	0	x	x	1	1	x
0	0	0	1	0	0	0	0	x	0	x	x	1
1	0	0	0	0	0	1	0	x	0	x	1	x
1	0	0	1	0	1	0	0	x	1	x	x	1
1	0	1	0	0	1	1	0	x	x	0	1	x
1	0	1	1	1	0	0	1	x	x	1	x	1
1	1	0	0	1	0	1	x	0	0	x	1	x
1	1	0	1	1	1	0	x	0	1	x	x	1
1	1	1	0	1	1	1	x	0	x	0	1	x
1	1	1	1	0	0	0	x	1	x	1	x	1

K-map Simplification:

For J_A

UD QA \ QB QC	00	01	11	10
00	1	0	0	0
01	X	X	X	X
11	X	X	X	X
10	0	0	1	0

$J_A = \overline{UD}\ \overline{QB}\ \overline{QC} + UD\ QB\ QC$

For K_A

UD QA \ QB QC	00	01	11	10
00	X	X	X	X
01	1	0	0	0
11	0	0	1	0
10	X	X	X	X

$K_A = \overline{UD}\ \overline{QB}\ \overline{QC} + UD\ QB\ QC$

For J_B

UD QA \ QB QC	00	01	11	10
00	1	0	X	X
01	1	0	X	X
11	0	1	X	X
10	0	1	X	X

$J_B = UD\ QC + \overline{UD}\ \overline{QC}$

For K_B

UD QA \ QB QC	00	01	11	10
00	X	X	0	1
01	X	X	0	1
11	X	X	1	0
10	X	X	1	0

$K_B = UD\ QC + \overline{UD}\ \overline{QC}$

For J_C

UD QA \ QB QC	00	01	11	10
00	1	X	X	1
01	1	X	X	1
11	1	X	X	1
10	1	X	X	1

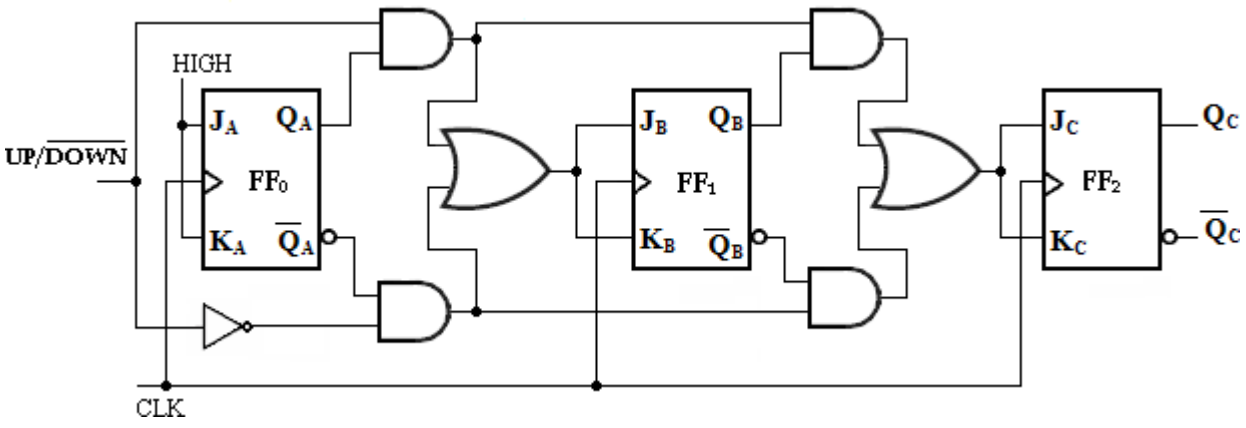
$J_C = 1$

For K_C

UD QA \ QB QC	00	01	11	10
00	X	1	1	X
01	X	1	1	X
11	X	1	1	X
10	X	1	1	X

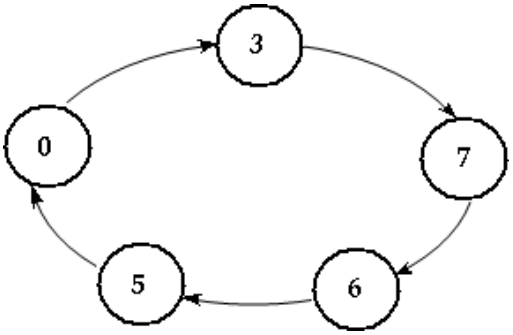
$K_C = 1$

Logic Diagram:



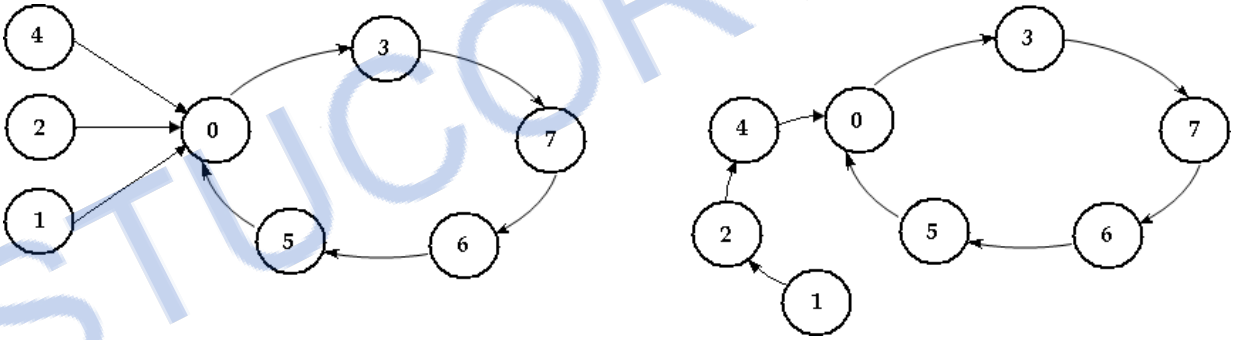
4.20 LOCKOUT CONDITION:

In a counter if the next state of some unused state is again a used state and if by chance the counter happens to find itself in the unused states and never arrived at a used state then the counter is said to be in the lockout condition.



Desired Sequence

The circuit that goes in lockout condition is called ***bushless*** circuit. To make sure that the counter will come to the initial state from any unused state, the additional logic circuit is necessary. To ensure that the lockout does not occur, the counter should be designed by forcing the next state to be the initial state from the unused states as shown below.



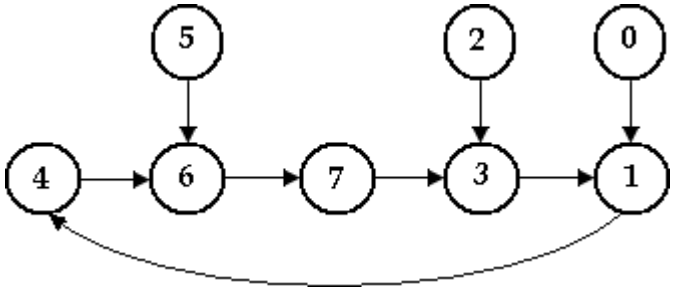
State diagram for removing lockout

8. Design a synchronous counter for

$4 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 1 \rightarrow 4 \ldots$ Avoid lockout condition. Use JK type design.

Soln:

State diagram:



Here, states 5, 2 and 0 are forced are forced to go into 6, 3 and 1state, respectively to avoid lockout condition.

Excitation table:

Excitation Table for JK 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 counter:

Present State			Next State			Flip-Flop Inputs					
Q_A	Q_B	Q_C	Q_{A+1}	Q_{B+1}	Q_{C+1}	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	x	0	x	1	x
0	0	1	1	0	0	1	x	0	x	x	1
0	1	0	0	1	1	0	x	x	0	1	x
0	1	1	0	0	1	0	x	x	1	x	0
1	0	0	1	1	0	x	0	1	x	0	x
1	0	1	1	1	0	x	0	1	x	x	1
1	1	0	1	1	1	x	0	x	0	1	x
1	1	1	0	1	1	x	1	x	0	x	0

K-map Simplification:

For J_A

Q_A \ Q_BQ_C	00	01	11	10
0	0	1	0	0
1	X	X	X	X

$J_A = \overline{Q}_B Q_C$

For K_A

Q_A \ Q_BQ_C	00	01	11	10
0	X	X	X	X
1	0	0	1	0

$K_A = Q_B Q_C$

For J_B

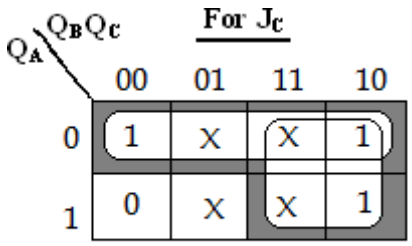
Q_A \ Q_BQ_C	00	01	11	10
0	0	0	X	X
1	1	1	X	X

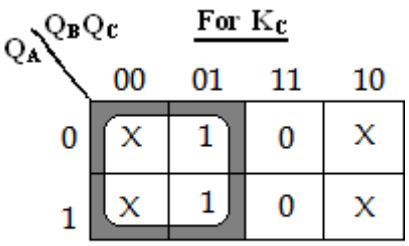
$J_B = Q_A$

For K_B

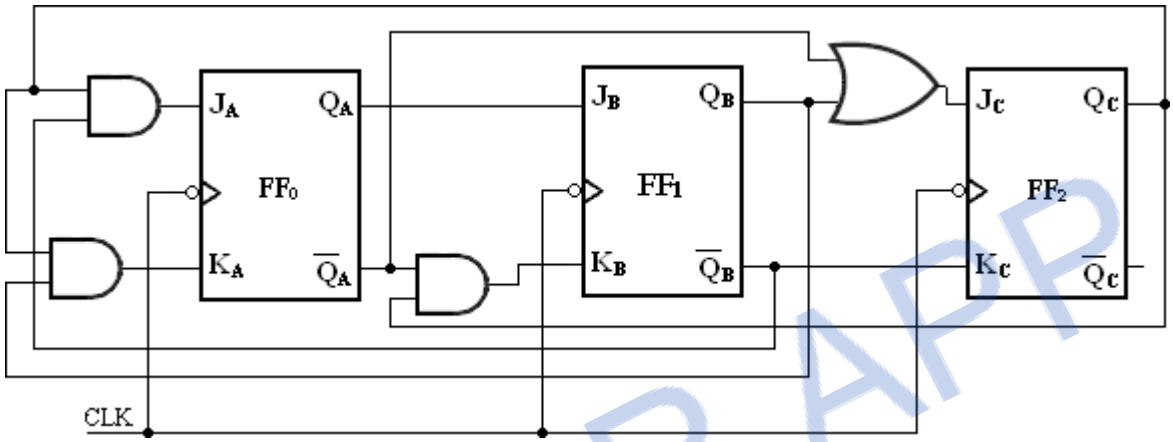
Q_A \ Q_BQ_C	00	01	11	10
0	X	X	1	0
1	X	X	0	0

$K_B = \overline{Q}_A Q_C$





Logic Diagram:



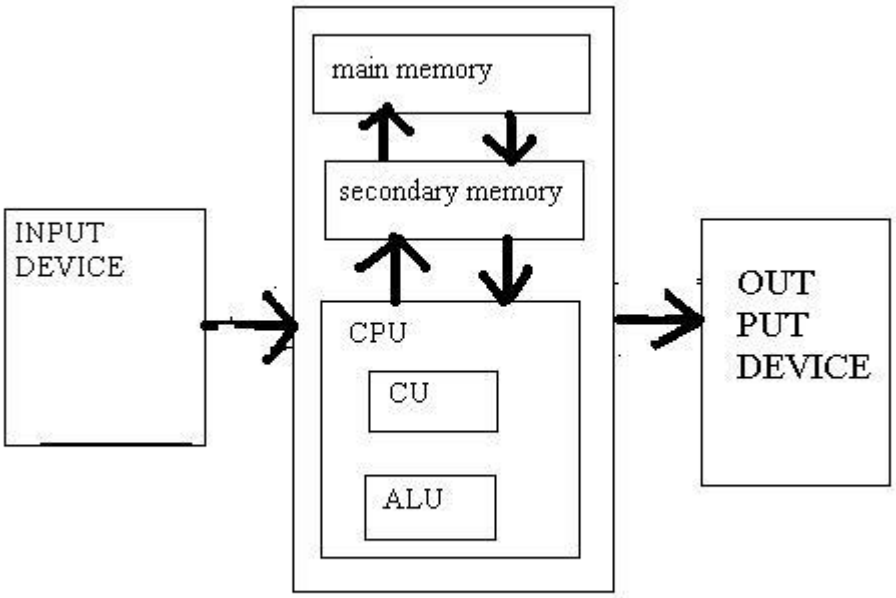
UNIT III

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.

3.1 Functional Units of Digital Computer

- A computer organization describes the functions and design of the various units of a digital system.
- A general-purpose computer system is the best-known example of a digital system. Other examples include telephone switching exchanges, digital voltmeters, digital counters, electronic calculators and digital displays.
- Computer architecture deals with the specification of the instruction set and the hardware units that implement the instructions.
- Computer hardware consists of electronic circuits, displays, magnetic and optic storage media and also the communication facilities.
- Functional units are a part of a CPU that performs the operations and calculations called for by the computer program.
- Functional units of a computer system are parts of the CPU (Central Processing Unit) that performs the operations and calculations called for by the computer program. A computer consists of five main components namely, Input unit, Central Processing Unit, Memory unit Arithmetic & logical unit, Control unit and an Output unit.



BLOCK DIAGRAM OF A DIGITAL COMPUTER

Input unit

- Input units are used by the computer to read the data. The most commonly used input devices are keyboards, mouse, joysticks, trackballs, microphones, etc.
- However, the most well-known input device is a keyboard. Whenever a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or the processor.

Central processing unit

- Central processing unit commonly known as CPU can be referred as an electronic circuitry within a computer that carries out the instructions given by a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

Memory unit

- The Memory unit can be referred to as the storage area in which programs are kept which are running, and that contains data needed by the running programs.
- The Memory unit can be categorized in two ways namely, primary memory and secondary memory.
- It enables a processor to access running execution applications and services that are temporarily stored in a specific memory location.

- Primary storage is the fastest memory that operates at electronic speeds. Primary memory contains a large number of semiconductor storage cells, capable of storing a bit of information. The word length of a computer is between 16-64 bits.
- It is also known as the volatile form of memory, means when the computer is shut down, anything contained in RAM is lost.
- Cache memory is also a kind of memory which is used to fetch the data very soon. They are highly coupled with the processor.
- The most common examples of primary memory are RAM and ROM.
- Secondary memory is used when a large amount of data and programs have to be stored for a long-term basis.
- It is also known as the Non-volatile memory form of memory, means the data is stored permanently irrespective of shut down.
- The most common examples of secondary memory are magnetic disks, magnetic tapes, and optical disks.

Arithmetic & logical unit

- Most of all the arithmetic and logical operations of a computer are executed in the ALU (Arithmetic and Logical Unit) of the processor. It performs arithmetic operations like addition, subtraction, multiplication, division and also the logical operations like AND, OR, NOT operations.

Control unit

- The control unit is a component of a computer's central processing unit that coordinates the operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.
- The control unit is also known as the nerve center of a computer system.
- Let's us consider an example of addition of two operands by the instruction given as Add LOCA, RO. This instruction adds the memory location LOCA to the operand in the register RO and places the sum in the register RO. This instruction internally performs several steps.

Output Unit

- The primary function of the output unit is to send the processed results to the user. Output devices display information in a way that the user can understand.
- Output devices are pieces of equipment that are used to generate information or any other response processed by the computer. These devices display information that has been held or generated within a computer.

- The most common example of an output device is a monitor.

3.2 Von-Neumann Model

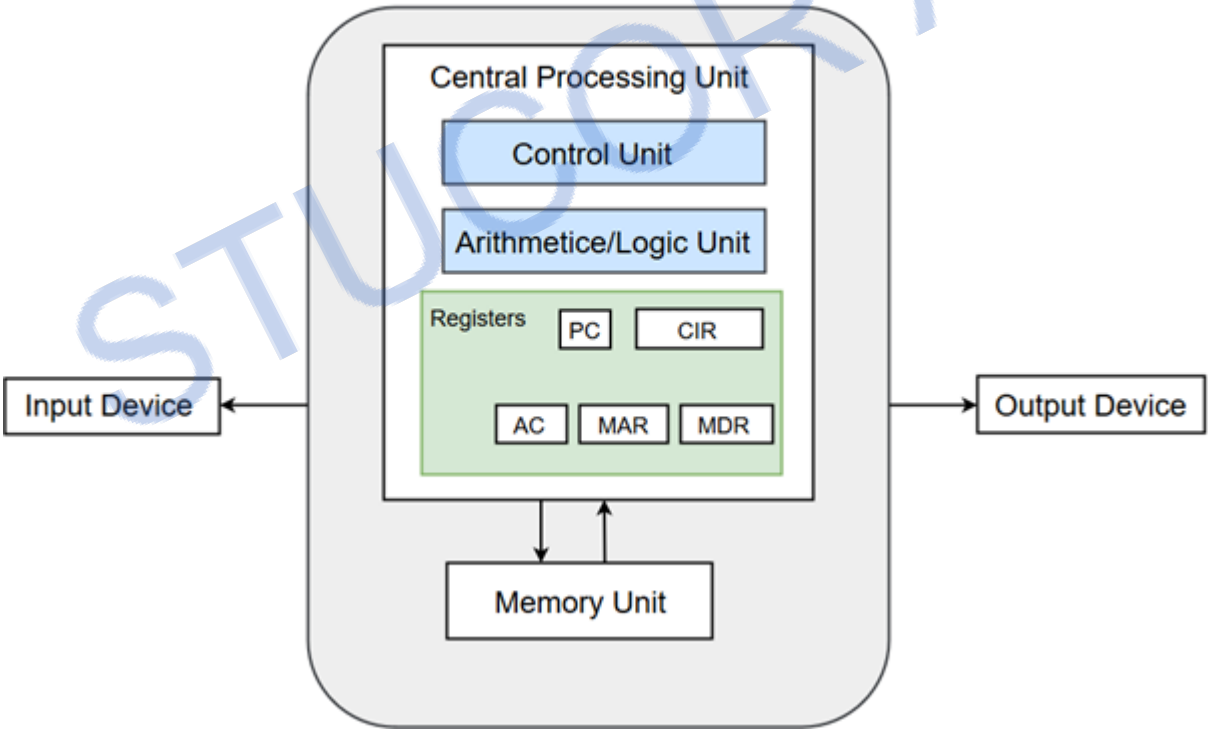
Von-Neumann proposed his computer architecture design in 1945 which was later known as Von-Neumann Architecture. It consisted of a Control Unit, Arithmetic, and Logical Memory Unit (ALU), Registers and Inputs/Outputs.

Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

A Von Neumann-based computer:

- Uses a single processor
- Uses one memory for both instructions and data.
- Executes programs following the fetch-decode-execute cycle

Von-Neumann Basic Structure:



Components of Von-Neumann Model:

- Central Processing Unit
- Buses
- Memory Unit

Central Processing Unit

The part of the Computer that performs the bulk of data processing operations is called the Central Processing Unit and is referred to as the CPU.

The Central Processing Unit can also be defined as an electric circuit responsible for executing the instructions of a computer program.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer.

The major components of CPU are Arithmetic and Logic Unit (ALU), Control Unit (CU) and a variety of registers.

Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU) performs the required micro-operations for executing the instructions. In simple words, ALU allows arithmetic (add, subtract, etc.) and logic (AND, OR, NOT, etc.) operations to be carried out.

Control Unit

The Control Unit of a computer system controls the operations of components like ALU, memory and input/output devices.

The Control Unit consists of a program counter that contains the address of the instructions to be fetched and an instruction register into which instructions are fetched from memory for execution.

Registers

Registers refer to high-speed storage areas in the CPU. The data processed by the CPU are fetched from the registers.

Following is the list of registers that plays a crucial role in data processing.

Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

Buses

Buses are the means by which information is shared between the registers in a multiple-register configuration system.

A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

Von-Neumann Architecture comprised of three major bus systems for data transfer.

Bus	Description
Address Bus	Address Bus carries the address of data (but not the data) between the processor and the memory.
Data Bus	Data Bus carries data between the processor, the memory unit and the input/output devices.
Control Bus	Control Bus carries signals/commands from the CPU.

Memory Unit

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the storage. The memory stores binary information in groups of bits called words. The internal structure of a memory unit is specified by the number of words it contains and the number of bits in each word.

Two major types of memories are used in computer systems:

1. RAM (Random Access Memory)
2. ROM (Read-Only Memory)

3.3 Operation and Operands of Computer Hardware Instruction

Computer instruction is a binary code that determines the micro-operations in a sequence for a computer. They are saved in the memory along with the information. Each computer has its specific group of instructions. They can be categorized into two elements as ***Operation codes*** (Opcodes) and ***Address***. Opcodes specify the operation for specific instructions, and an address determines the registers or the areas used for that operation.

Operands are definite elements of computer instruction that show what information is to be operated on. The most important general categories of data are

1. Addresses
2. Numbers
3. Characters
4. Logical data

In many cases, some calculation must be performed on the operand reference to determine the main or virtual memory address.

In this context, addresses can be considered to be unsigned integers. Other common data types are numbers, characters, and logical data, and each of these is briefly described below. Some machines define specialized data types or data structures. For example, machine operations may operate directly on a list or a string of characters.

Addresses

Addresses are nothing but a form of data. Here some calculations must be performed on the operand reference in an instruction, which is to determine the physical address of an instruction.

Numbers

All machine languages include numeric data types. Even in non-numeric data processing, numbers are needed to act as counters, field widths, etc. An important difference between numbers used in ordinary mathematics and numbers stored in a computer is that the latter is limited. Thus, the programmer is faced with understanding the consequences of rounding, overflow and underflow.

Here are the three types of numerical data in computers, such as:

1. Integer or fixed point: Fixed point representation is used to store integers, the positive and negative whole numbers (… -3, -2, -1, 0, 1, 2, 3, …). However, the programmer assigns a radix point location to each number and tracks the radix point through every operation. High-level programs, such as C and BASIC usually allocate 16 bits to store each integer. Each fixed point binary number has three important parameters that describe it:

- Whether the number is signed or unsigned,
- The position of the radix point to the right side of the sign bit (for signed numbers), or the position of the radix point to the most significant bit (for unsigned numbers).
- And the number of fractional bits stored.

2. Floating point: A Floating Point number usually has a decimal point, which means **0, 3.14, 6.5,** and**-125.5** are Floating Point

The term *floating point* is derived from the fact that there is no fixed number of digits before and after the decimal point, which means the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called ***fixed-point*** representations. In general, floating-point representations are slower and less accurate than fixed-point representations, but they can handle a larger range of numbers.

3. Decimal number: The decimals are an extension of our number system. We also know that decimals can be considered fractions with 10, 100, 1000, etc. The numbers expressed in the decimal form are called decimal numbersor decimals. For example:1, 4.09, 13.83, etc. A decimal number has two parts, and a dot separates these parts (.) called the ***decimal point***.

- **Whole number part:** The digits lying to the left of the decimal point form the whole number part. The places begin with ones, tens, hundreds, thousands and so on.
- **Decimal part:** The decimal point and the digits laying on the right of the decimal point form the decimal part. The places begin with tenths, hundredths, thousandths and so on.

Characters

A common form of data is text or character strings. While textual data are most convenient for humans. But computers work in binary. So, all characters whether letters, punctuation or digits, are

stored as binary numbers. All of the characters that a computer can use are called ***character set***s. Here are the two common standards, such as:

1. American Standard Code for Information Interchange (ASCII)
2. Unicode

ASCII uses seven bits, giving a character set of 128 characters. The characters are represented in a table called the ASCII table. The 128 characters include:

- 32 control codes (mainly to do with printing)
- 32 punctuation codes, symbols, and space
- 26 upper-case letters
- 26 lower-case letters
- numeric digits 0-9

We can say that the letter 'A' is the first letter of the alphabet; 'B' is the second, and so on, all the way up to 'Z', which is the 26th letter. In ASCII, each character has its own assigned number. Denary, binary and hexadecimal representations of ASCII characters are shown in the below table.

Character	Denary	Binary	Hexadecimal
A	65	1000001	41
Z	90	1011010	5A
a	97	1100001	61
z	122	1111010	7A
0	48	0110000	30
9	57	0111001	39
Space	32	0100000	20
!	33	0100001	21

A is represented by the denary number 65 (binary 1000001, hexadecimal 41), B by 66 (binary 1000010, hexadecimal 42) and so on up to Z, which is represented by the denary number 90 (binary 1011010, hexadecimal 5A).

Similarly, lower-case letters start at denary 97 (binary 1100001, hexadecimal 61) and end at denary 122 (binary 1111010, hexadecimal 7A). When data is stored or transmitted, its ASCII or Unicode number is used, not the character itself.

For example, the word "Computer" would be represented as:

1000011 1101111 1101101 1110000 1110101 1110100 1100101 1110010

On the other hand, ***IRA*** is also widely used outside the United States. A unique 7-bit pattern represents each character in this code. Thus, 128 different characters can be represented, and more than necessary to represent printable characters, and some of the patterns represent control

characters. Some control characters control the printing of characters on a page, and others are concerned with communications procedures.

IRA-encoded characters are always stored and transmitted using 8 bits per character. The 8 bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Logical data

Normally, each word or other addressable unit (byte, half-word, and so on) is treated as a single unit of data. Sometimes, it is useful to consider an n-bit unit consisting of 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be logical data.

The ***Boolean*** data can only represent two values: true or false. Although only two values are possible, they are rarely implemented as a single binary digit for efficiency reasons. Many programming languages do not have an explicit Boolean type, instead of interpreting 0 as false and other values as true. Boolean data refers to the logical structure of how the language is interpreted to the machine language. In this case, a Boolean 0 refers to the logic False, and true is always a non zero, especially one known as Boolean 1.

There are two advantages to the bit-oriented view:

- We may want to store an array of Boolean or binary data items, in which each item can take on only the values 0 and 1. With logical data, memory can be used most efficiently for this storage.
- There are occasions when we want to manipulate the bits of a data item.

3.4 Instruction set Architecture(ISA):

An Instruction Set Architecture (ISA) is **part of the abstract model of a computer that defines how the CPU is controlled by the software**. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done

Two types of instruction set architectures are

The two main categories of instruction set architectures, **CISC (such as Intel's x86 series) and RISC (such as ARM and MIPS)**,

The ISA of a processor can be described using 5 catagories:

Operand Storage in the CPU
Number of explicit named operands
Operand location
Operations
Type and size of operands

The 3 most common types of ISAs are:

1. ***Stack*** - The operands are implicitly on top of the stack.
2. ***Accumulator*** - One operand is implicitly the accumulator.
3. *General Purpose Register (GPR)* - All operands are explicitely mentioned, they are either registers or memory locations

Stack	Accumulator	GPR
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE R1,C
POP C	-	-

The i8086 has many instructions that use implicit operands although it has a general register set. The i8051 is another example, it has 4 banks of GPRs but most instructions must have the A register as one of its operands.

Stack

Advantages: Simple Model of expression evaluation (reverse polish). Short instructions.
Disadvantages: A stack can't be randomly accessed This makes it hard to generate eficient code. The stack itself is accessed every operation and becomes a bottleneck.

Accumulator

Advantages: Short instructions.
Disadvantages: The accumulator is only temporary storage so memory traffic is the highest for this approach.

GPR

Advantages: Makes code generation easy. Data can be stored for long periods in registers.
Disadvantages: All operands must be named leading to longer instructions.

Reduced Instruction Set Computer (RISC):

RISC stands for Reduced Instruction Set Computer. The ISA is composed of instructions that all have exactly the same size, usualy 32 bits. Thus they can be pre-fetched and pipelined succesfuly. All ALU instructions have 3 operands which are only registers. The only memory access is through explicit LOAD/STORE instructions.
Thus C = A + B will be assembled as:

```
LOAD  R1,A
LOAD  R2,B
ADD   R3,R1,R2
STORE C,R3
```

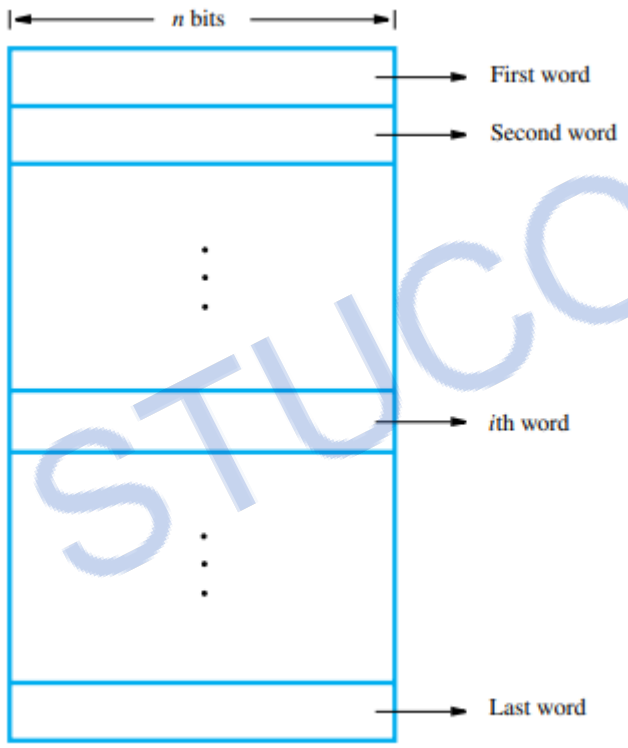
Although it takes 4 instructions we can reuse the values in the registers.

Complex Instruction Set Architecture (CISC) :
The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex

Memory Locations and Addresses

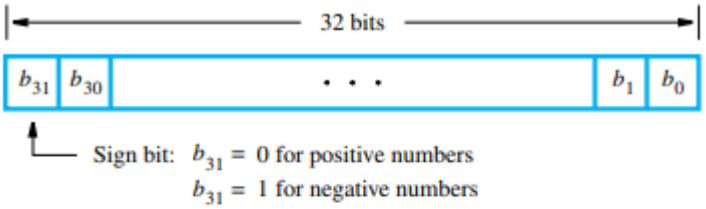
The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually.

The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. The memory of a computer can be schematically represented as a collection of words.

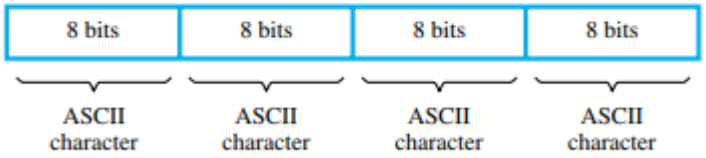


Memory words.

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure



(a) A signed integer



(b) Four characters

Examples of encoded information in a 32-bit word.

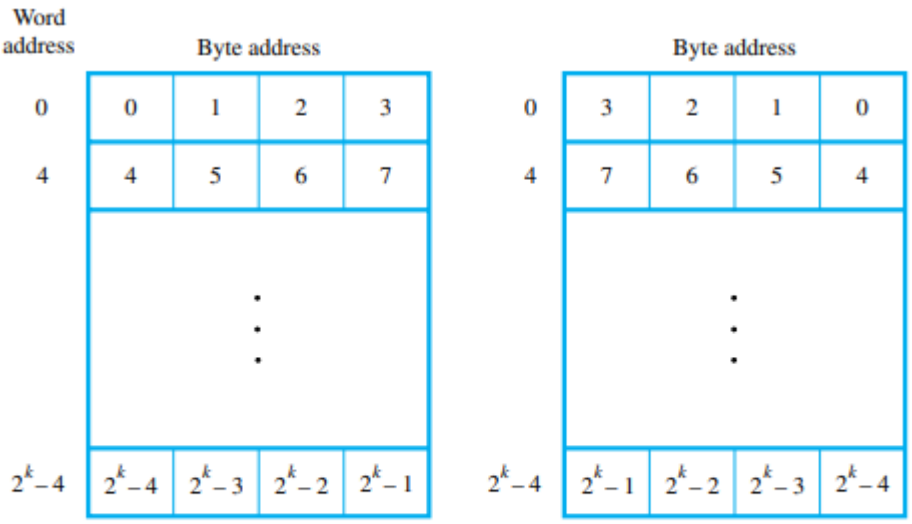
A unit of 8 bits is called a byte. Machine instructions may require one or more words for their representation.

After we have described instructions at the assembly-language level. Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each location. It is customary to use numbers from 0 to 2k − 1, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2k addressable locations. The 2k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 224 (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 220 (1,048,576). A 32-bit address creates an address space of 232 or 4G (4 giga) locations, where 1G is 230. Other notational conventions that are commonly used are K (kilo) for the number 210 (1,024), and T (tera) for the number 240

Byte Addressability :

A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term byte-addressable memory is used for this assignment. Byte locations have addresses 0, 1, 2,.... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes.

There are two ways that byte addresses can be assigned across words **big-endian** and **Little endian**



(a) Big-endian assignment (b) Little-endian assignment

Byte and word addressing.

The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word.

The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words "more significant" and "less significant" are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8,..., are taken as the addresses of successive words in the memory of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

Memory Operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor.

Thus, two basic operations involving the memory are needed, , namely Read and Write.

Read Operation:

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

Write Operation:

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

3.5 Instructions and Instruction Sequencing

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen.

A computer must have instructions capable of performing four types of operations:

• Data transfers between the memory and the processor registers

• Arithmetic and logic operations on data

• Program sequencing and control

• I/O transfers

We begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation

Register Transfer Notation

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem.

Example 1:

R2 ← [LOC]

This expressionmeans that the contents of memory location LOC are transferred into processor register R2

Example 2:

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

R4 ← [R2]+[R3]

This type of notation is known as Register Transfer Notation (RTN). Note that the righthand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location

Assembly-Language Notation:

to represent machine instructions and programs we use assembly –Language notation

Example 1:

Load R2, LOC

a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are loaded into a processor register

Example 2:

Add R4, R2, R3

adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement, registers R2 and R3 hold the source operands, while R4 is the destination

RISC and CISC Instruction Sets

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in "pipelined" fashion to overlap activity and reduce total execution time of a program. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called Reduced Instruction Set Computers (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called Complex Instruction Set Computers (CISC).

Introduction to RISC Instruction Sets:

Two key characteristics of RISC instruction sets are: • Each instruction fits in a single word. • A load/store architecture is used, in which – Memory operands are accessed only using Load and Store instructions. – All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer. Processor registers do not contain valid operands at that time . If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers. This task is done by Load instructions which copy the contents of a memory location into a processor register. Load instructions are of the form

Load destination, source

Or more specifically

Load processor_register, memory_location

Example:

The operation of adding two numbers is a fundamental capability in any computer.

The statement C = A + B

The required action can be accomplished by a sequence of simple machine instructions. We choose to use registers R2, R3, and R4 to perform the task with four instructions:

Load R2, A

Load R3, B

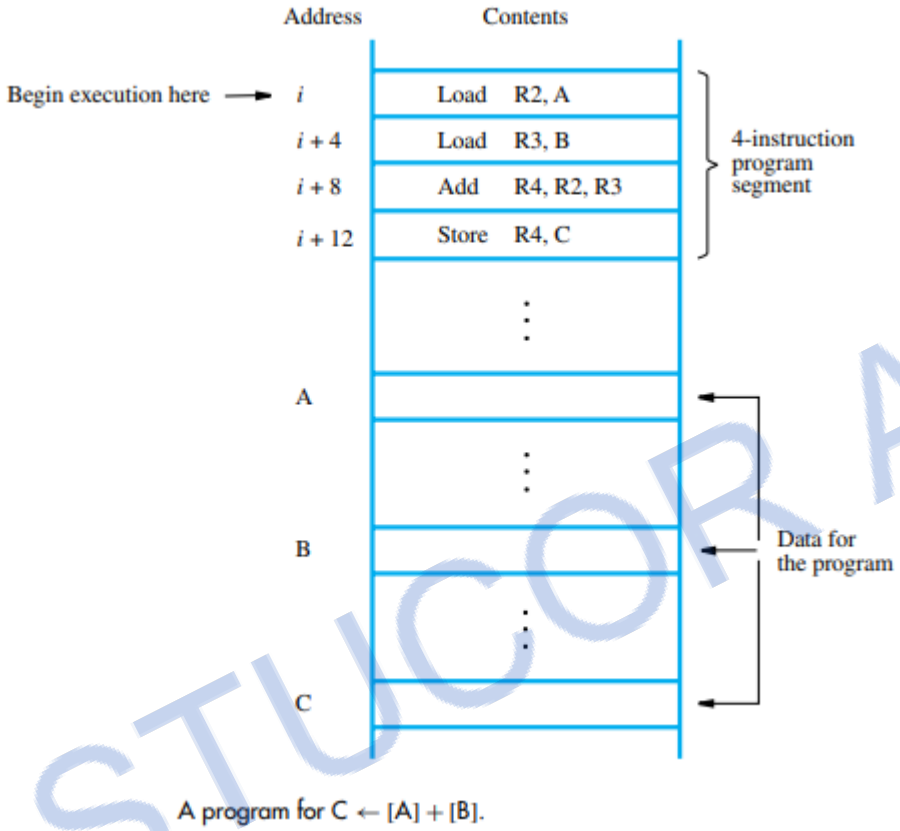
Add R4, R2, R3

Store R4, C

Instruction Execution and Straight-Line Sequencing:

we used the task C = A + B

implemented as C ← [A] + [B]



A program for C ← [A] + [B].

We assume that the word length is 32 bits and the memory is byte-addressable. The four instructions of the program are in successive word locations, starting at location i. Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses i + 4, i + 8, and i + 12. For simplicity, we assume that a desired memory address can be directly specified in Load and Store instructions, although this is not possible if a full 32-bit address is involved.

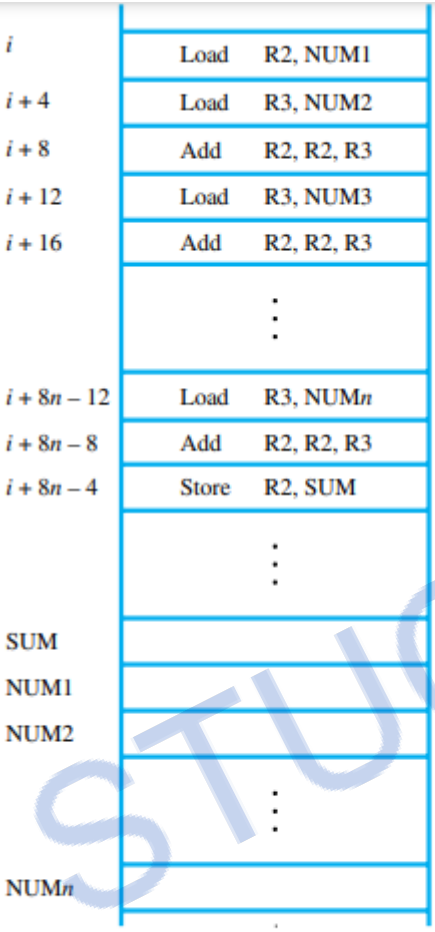
Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location i + 12 is executed, the PC contains the value i + 16, which is the address of the first instruction of the next program segment. Executing a given instruction is a two-phase procedure. In the first phase, called instruction fetch, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the instruction register (IR) in the processor. At the start of the second phase, called instruction execute, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to

point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

Branching:

Consider the task of adding a list of n number

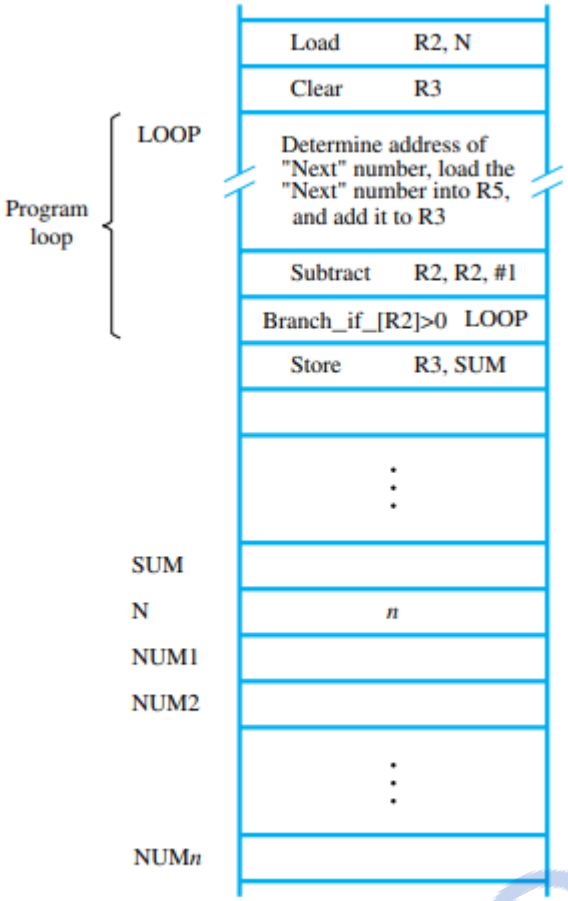
The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2,..., NUMn, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.



A program for adding *n* numbers.

Instead of using a long list of Load and Add instructions,

it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch_if_[R2]>0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways, as will be described in Section 2.4. For now, we concentrate on how to create and control a program loop. Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction.



Using a loop to add *n* numbers.

Subtract R2, R2, #1

reduces the contents of R2 by 1 each time through the loop. Execution of the loop is repeated as long as the contents of R2 are greater than zero. We now introduce branch instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order. A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

Branch_if_[R2]>0 LOOP

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the nth pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

Encoding of Machine Instructions:

we have introduced a variety of useful instructions and addressing modes. We have used a generic form of assembly language to emphasize basic concepts without relying on processor-specific acronyms or mnemonics. Assembly-language instructions symbolically express the actions that must be performed by the processor circuitry.

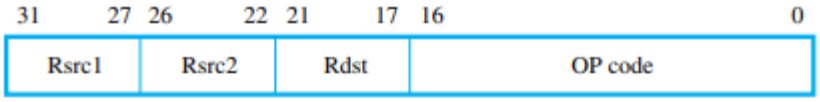
To be executed in a processor, assembly-language instructions must be converted by the assembler program, into machine instructions that are encoded in a compact binary pattern.

Let us now examine how machine instructions may be formed.

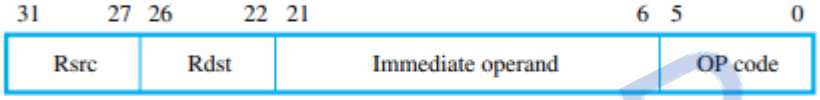
The Add instruction

Add Rdst, Rsrc1, Rsrc2

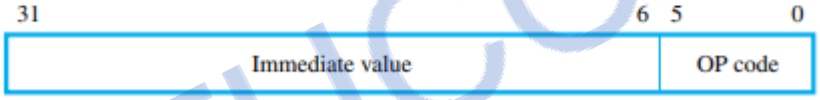
The above instruction representative of a class of three-operand instructions that use operands in processor registers. Registers Rdst, Rsrc1, and Rsrc2 hold the destination and two source operands. If a processor has 32 registers, then it is necessary to use five bits to specify each of the three registers in such instructions. If each instruction is implemented in a 32-bit word, the remaining 17 bits can be used to specify the OP code that indicates the operation to be performed



(a) Register-operand format



(b) Immediate-operand format



(c) Call format

Possible instruction formats.

Now consider instructions in which one operand is given using the Immediate addressing mode, such as Add Rdst, Rsrc, #Value

Of the 32 bits available, ten bits are needed to specify the two registers. The remaining 22 bits must give the OP code and the value of the immediate operand. The most useful sizes of immediate operands are 32, 16, and 8 bits. Since 32 bits are not available, a good choice is to allocate 16 bits for the immediate operand. This leaves six bits for specifying the OP code.

This format can also be used for Load and Store instructions, where the Index addressing mode uses the 16-bit field to specify the offset that is added to the contents of the index register.

The format in Figure b can also be used to encode the Branch instructions. The Branch-greater-than instruction at memory address 128.

BGT R2, R0, LOOP

if the contents of register R0 are zero. The registers R2 and R0 can be specified in the two register fields in Figure b. The six-bit OP code has to identify the BGT operation. The 16-bit immediate field can be used to provide the information needed to determine the branch target address, which is the location of the instruction with the label LOOP. The target address generally comprises 32 bits. Since there is no space for 32 bits, the BGT instruction makes use of the immediate field to give an offset from the location of this instruction in the

program to the required branch target. At the time the BGT instruction is being executed, the program counter, PC, has been incremented to point to the next instruction, which is the Store instruction at address 132. Therefore, the branch offset is 132 − 112 = 20. Since the processor computes the target address by adding the current contents of the PC and the branch offset, the required offset in this example is negative, namely −20. Finally, we should consider the Call instruction, which is used to call a subroutine. It only needs to specify the OP code and an immediate value that is used to determine the address of the first instruction in the subroutine. If six bits are used for the OP code, then the remaining 26 bits can be used to denote the immediate value. This gives the format shown in c.

Addressing Modes:

The operation field of an instruction specifies the operation to be performed. And this operation must be performed on some data. So each instruction need to specify data on which the operation is to be performed. But the operand(data) may be in accumulator, general purpose register or at some specified memory location. So, appropriate location

(address) of data is need to be specified, and in computer, there are various ways of specifying the address of data. These various ways of specifying the address of data are known as “Addressing Modes”

So Addressing Modes can be defined as “The technique for specifying the address of the operands “ And in computer the address of operand i.e., the address where operand is actually found is known as **“Effective Address”.** Now, in addition to this, the two most prominent reason of why addressing modes are so important are:

First, the way the operand data are chosen during program execution is dependent on theaddressing mode of the instruction.

Second, the address field(or fields) in a typical instruction format are relatively small and sometimes we would like to be able to reference a large range of locations, so here to achieve this objective i.e., to fit this large range of location in address field, a variety of addressing techniques has been employed. As they reduce the number of field in the addressing field of the instruction.

Thus, Addressing Modes are very vital in Instruction Set Architecture(ISA).some notations are

A= Contents of an address field in the instruction

R= Contents of an address field in the instruction that refers to a register

EA= Effective Address(Actual address) of location containing the referenced operand.(X)= Contents of memory location x or register X.

Types Of Addressing Modes

Various types of addressing modes are:

1. Implied and Immediate Addressing Modes
2. Direct or Indirect Addressing Modes
3. Register Addressing Modes
4. Register Indirect Addressing Mode
5. Auto-Increment and Auto-Decrement Addressing Modes
6. Displacement Based Addressing Modes

1. Implied and Immediate Addressing Modes:

Implied Addressing Mode:

Implied Addressing Mode also known as "Implicit" or "Inherent“ addressing mode is the addressing mode in which, no operand(register or memory location or data) is specified in the instruction. As in this mode the operand are specified implicit in the definition of instruction.

“Complement Accumulator” is an Implied Mode instruction because the operand in the accumulator register is implied in the definition of instruction. In assembly language it is written as:

CMA: Take complement of content of AC Similarly, the instruction,

RLC: Rotate the content of Accumulator is an implied mode instruction.

All Register-Reference instruction that use an accumulator and Zero-Address instruction in a Stack Organised Computer are implied mode instructions, because in Register reference operand implied in accumulator and in Zero-Address instruction, the operand implied on the Top of Stack.

Immediate Addressing Mode:

In Immediate Addressing Mode operand is specified in the instruction itself. In other words, an immediate mode instruction has an operand field rather than an address field, which contain actual operand to be used in conjunction with the operand specified in the instruction. That is, in this mode, the format of instruction is:

As an example: The Instruction:

MVI 06 Move 06 to the accumulator

ADD 05 ADD 05 to the content of accumulator

- One of the operand is mentioned directly.
- Data is available as a part of instruction.
- Data is 8 0r 16 bit long.
- No memory reference is needed to fetch data

Immediate Mode :Eg.

Example 1 :

MOV CL, 03H

03 – 8 bit immediate source operand

CL – 8 bit register destination operand

Example 2:

ADD AX, 0525H

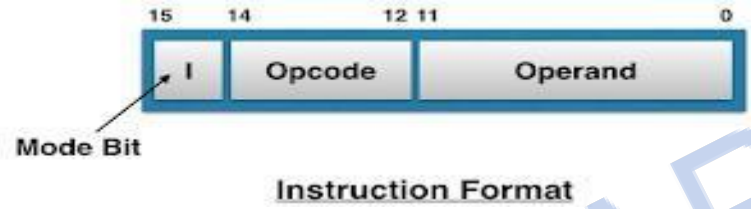
0525 – 16 bit immediate source operand

AX – 16 bit register destination operand.

2. Direct and Indirect Addressing Modes

The instruction format for direct and indirect addressing mode is shown below:

It consists of 3-bit opcode, 12-bit address and a mode bit designated as(I).**The mode bit (I) is zero for Direct Address and 1 for Indirect Address**. Now we will discuss about each in detail one by one.



Instruction Format

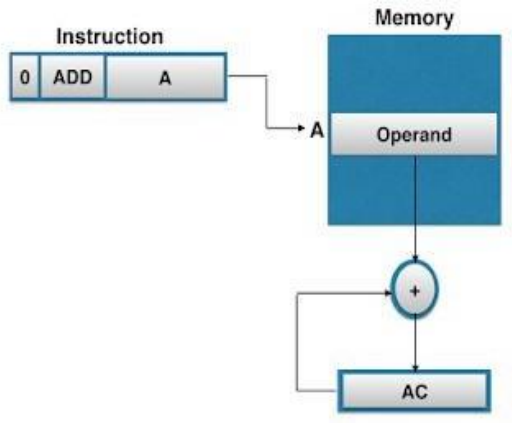
Direct Addressing Mode

Direct Addressing Mode is also known as "Absolute Addressing Mode". In this mode the address of data(operand) is specified in the instruction itself. That is, in this type of mode, the operand resides in memory and its address is given directly by the address field of the instruction. Means, in other words, in this mode, the address field contain the Effective Address of operand i.e., EA=A

As an example: Consider the instruction:

ADD A Means add contents of cell A to accumulator .

It Would look like as shown below:



Here, we see that in it Memory Address=Operand.

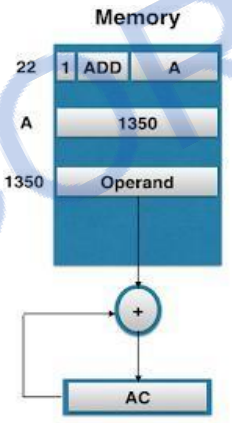
Indirect Addressing Mode:

In this mode, the address field of instruction gives the memory address where on, the operand is stored in memory. That is, in this mode, the address field of the instruction gives the address where the "Effective Address" is stored in memory. i.e., EA=(A)

Means, here, Control fetches the instruction from memory and then uses its address part to access memory again to read Effective Address.

As an example: Consider the instruction:

ADD (A) Means adds the content of cell pointed to contents of A to Accumulator. It look like as shown in figure below:



Thus in it, AC <-- M[M[A]]

[M=Memory]

i.e., (A)=1350=EA

3. Register Addressing Mode:

In Register Addressing Mode, the operands are in registers that reside within the CPU. That is, in this mode, instruction specifies a register in CPU, which contain the operand. It is like Direct Addressing Mode, the only difference is that the address field refers to a register instead of memory location.

i.e., EA=R

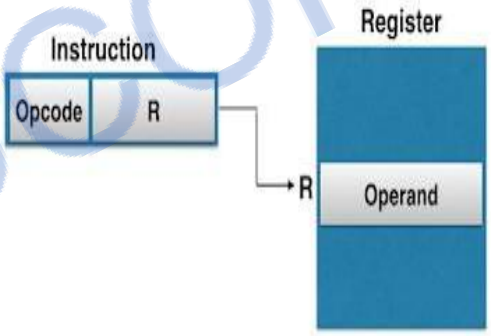
It look like as:

Example of such instructions are:

MOV AX, BX	Move contents of Register BX to AX
ADD AX, BX	Add the contents of register BX to AX

Here, AX, BX are used as register names which is of 16-bit register.

Thus, for a Register Addressing Mode, there is no need to compute the actual address as the operand is in a register and to get operand there is no memory access involved



4. Register Indirect Addressing Mode:

In Register Indirect Addressing Mode, the instruction specifies a register in CPU whose contents give the operand in memory. In other words, the selected register contain the address of operand rather than the operand itself. That is,

i.e., EA=(R)

Means, control fetches instruction from memory and then uses its address to access Register and looks in Register(R) for effective address of operand in memory.

It look like as:

Here, the parentheses are to be interpreted as meaning contents of.

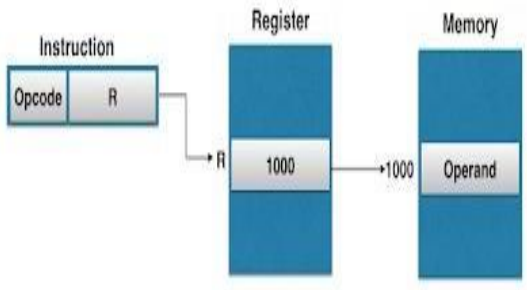
Example of such instructions are:

MOV AL, [BX]

Code example in Register:

MOV BX, 1000H

MOV 1000H, operand



From above example, it is clear that, the instruction(MOV AL, [BX]) specifies a register[BX], and in coding of register, we see that, when we move register [BX], the register contain the address of operand(1000H) rather than address itself.

5. Auto-increment and Auto-decrement Addressing Modes

These are similar to Register indirect Addressing Mode except that the register is incremented or decremented after(or before) its value is used to access memory. These modes are required because when the address stored in register refers to a table of data in memory, then it is necessary to increment or decrement the register after every access to table so that next value is accessed from memory.

Thus, these addressing modes are common requirements in computer.

Auto-increment Addressing Mode:

Auto-increment Addressing Mode are similar to Register Indirect Addressing Mode except that the register is incremented after its value is loaded (or accessed) at another location like accumulator(AC).

That is, in this case also, the Effective Address is equal to

EA=(R)

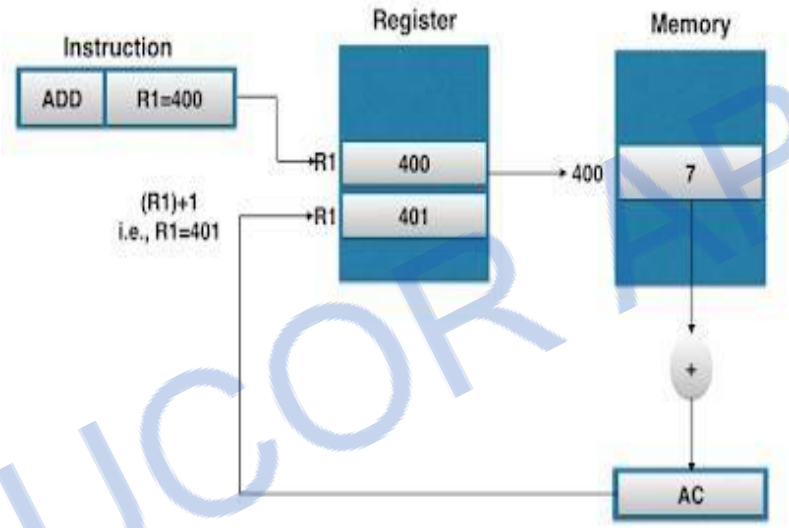
But, after accessing operand, register is incremented by 1.

As an example:

It look like as shown below:

Here, we see that effective address is (R)=400 and operand in AC is 7. And after loading R1 is incremented by 1.It becomes 401.

Means, here we see that, in the Auto-increment mode, the R1 register is increment to 401 after execution of instruction.



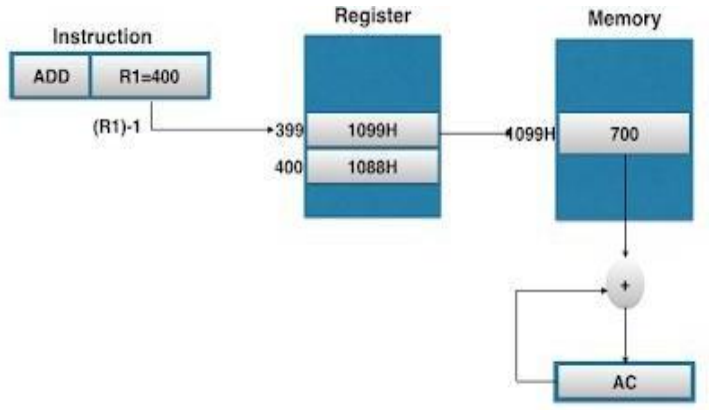
Auto-decrement Addressing Mode:

Auto-decrement Addressing Mode is reverse of auto-increment , as in it the register is decrement before the execution of the instruction. That is, in this case, effective address is equal to

EA=(R) - 1

As an example:

It look like as shown below:



Here, we see that, in the Auto-decrement mode, the register R1 is decremented to 399 prior to execution of the instruction, means the operand is loaded to accumulator, is of address 1099H in memory, instead of 1088H.Thus, in this case effective address is 1099H and contents loaded into accumulator is 700.

6. Displacement Based Addressing Modes:

Displacement Based Addressing Modes is a powerful addressing mode as it is a combination of direct addressing or register indirect addressing mode. i.e., EA=A+(R)

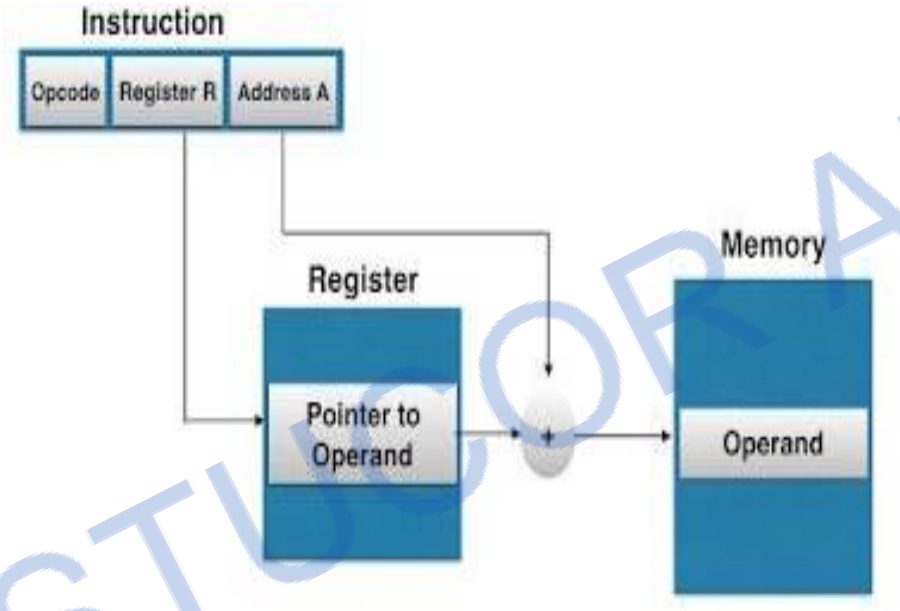
Means, Displacement Addressing Modes requires that the instruction have two address fields, at least one of which is explicit means, one is address field indicate direct address and other indicate indirect address.

That is, value contained in one addressing field is A, which is used directly and the value in other address field is R, which refers to a register whose contents are to be added to produce effective address.

There are three areas where Displacement Addressing modes are used. In other words, Displacement Based Addressing Modes are of three types. These are:

1. Relative Addressing Mode
2. Base Register Addressing Mode
3. Indexing Addressing Mode

Now we will explore to each one by one.



1. Relative Addressing Mode:

In Relative Addressing Mode , the contents of program counter is added to the address part of instruction to obtain the Effective Address.

That is, in Relative Addressing Mode, the address field of the instruction is added to implicitly reference register Program Counter to obtain effective address.

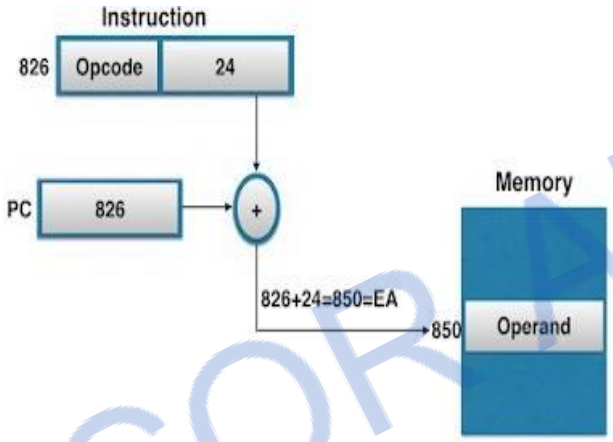
i.e., EA=A+PC

It becomes clear with an example:

Assume that PC contains the no.- 825 and the address part of instruction contain the no.- 24, then the instruction at location 825 is read from memory during fetch phase and the Program Counter is then incremented by one to 826.

The effective address computation for relative address mode is 26+24=850

Thus, Effective Address is displacement relative to the address of instruction. Relative Addressing is often used with branch type instruction



2. Index Register Addressing Mode

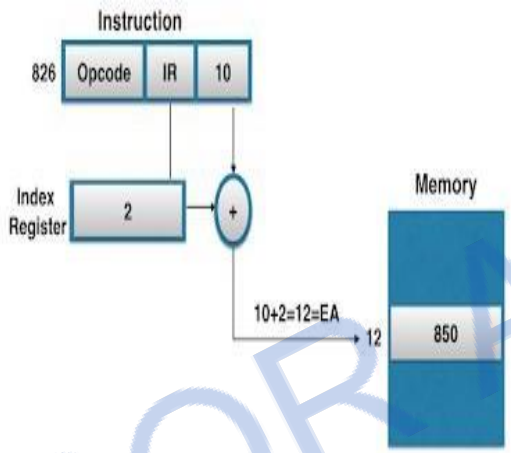
In indexed addressing mode, the content of Index Register is added to direct address part(or field) of instruction to obtain the effective address. Means, in it, the register indirect addressing field of instruction point to Index Register, which is a special CPU register that contain an Indexed value, and direct addressing field contain base address.

As, indexed type instruction make sense that data array is in memory and each operand in the array is stored in memory relative to base address. And the distance between the beginning address and the address of operand is the indexed value stored in indexed register.

Any operand in the array can be accessed with the same instruction, which provided that the index register contains the correct index value i.e., the index register can be incremented to facilitate access to consecutive operands.

Thus, in index addressing mode

EA=A+Index



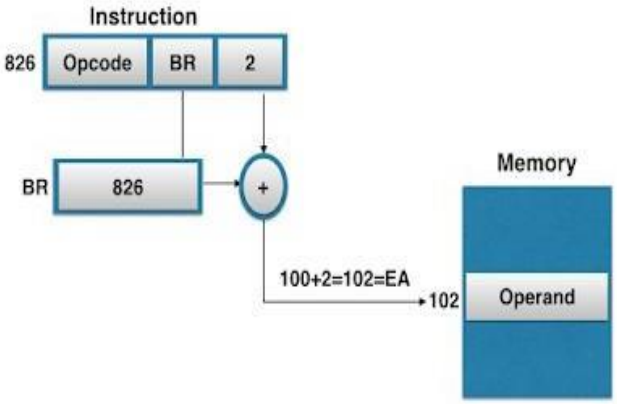
3. Base Register Addressing Mode:

In this mode, the content of the Base Register is added to the direct address part of the instruction to obtain the effective address.

Means, in it the register indirect address field point to the Base Register and to obtain EA, the contents of Instruction Register, is added to direct address part of the instruction.

This is similar to indexed addressing mode except that the register is now called as Base Register instead of Index Register.

That is, the EA=A+Base



Thus, the difference between Base and Index mode is in the way they are used rather than the way they are computed. An Index Register is assumed to hold an index number that is relative to the address part of the instruction. And a Base Register is assumed to hold a base address and thedirect address field of instruction gives a displacement relative to this base address.

Thus, the Base register addressing mode is used in computer to facilitate the relocation of programs in memory. Means, when programs and data are moved from one segment of memory to another, then Base address is changed, the displacement value of instruction do not change.

So, only the value of Base Register requires updation to reflect the beginning of new memory segment.

UNIT IV PROCESSOR

Instruction Execution

The execution of an instruction in a processor can be split up into a number of stages. How many stages there are, and the purpose of each stage is different for each processor design. Examples includes 2 stages (Instruction Fetch / Instruction Execute) and 3 stages (Instruction Fetch, Instruction Decode, Instruction Execute).

The MIPS processor has 5 stages:

IF The Instruction Fetch stage fetches the next instruction from memory using the address in the PC (Program Counter) register and stores this instruction in the IR (Instruction Register)

ID The Instruction Decode stage decodes the instruction in the IR, calculates the next PC, and reads any operands required from the register file.

EX The Execute stage "executes" the instruction. In fact, all ALU operations are done in this stage. (The ALU is the Arithmetic and Logic Unit and performs operations such as addition, subtraction, shifts left and right, etc.)

MA The Memory Access stage performs any memory access required by the current instruction, So, for loads, it would load an operand from memory. For stores, it would store an operand into memory. For all other instructions, it would do nothing.

WB For instructions that have a result (a destination register), the Write Back writes this result back to the register file. Note that this includes nearly all instructions, except nops (a nop, no-op or no-operation instruction simply does nothing) and s (stores).

BASIC MIPS IMPLEMENTATION

MIPS implementation 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

This subset does not include all the integer instructions (for example, shift,multiply, and divide are missing), nor does it include any floating-point instructions.

An Overview of the Implementation

The core MIPS instructions includes

- The integer arithmetic-logical instructions,
- The memory-reference instructions, and
- The branch instructions.

For every instruction, the first two steps are identical:

1. Send the *program counter* (PC) to the memory that contains the code andfetch the instruction from that memory.

2. Read one or two registers, using fields of the instruction to select the registersto read. For the load word instruction, we need to read only one register, butmost other instructions require reading two registers.

- ❖ For example, 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 operationexecution, and branches for comparison.
- ❖ 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.

Figure 3.1 shows the high-level view of a MIPS implementation, focusing onthe various functional units and their interconnection.

- ❖ A logic element need to be added that chooses from among the multiple sources and steers one of those sources to its destination.
- ❖ This selection is commonly done with a device called a ***multiplexor***, although this devicemight better be called a ***data selector***.
- ❖ The multiplexor selects from among several inputs based on the setting of its control lines. The control lines are set based primarily on information taken from the instruction being executed.

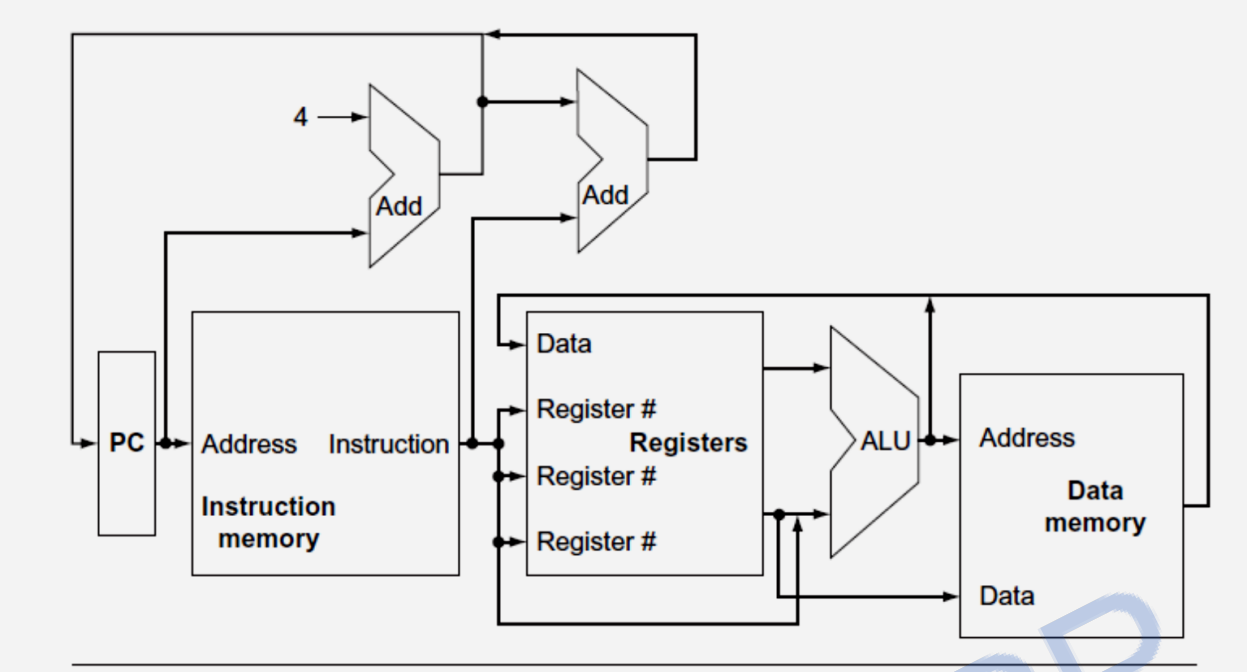


Fig 3.1: An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

Figure 3.2 shows the datapath of Figure 3.1 with the three required multiplexors added, as well as control lines for the major functional units. 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.

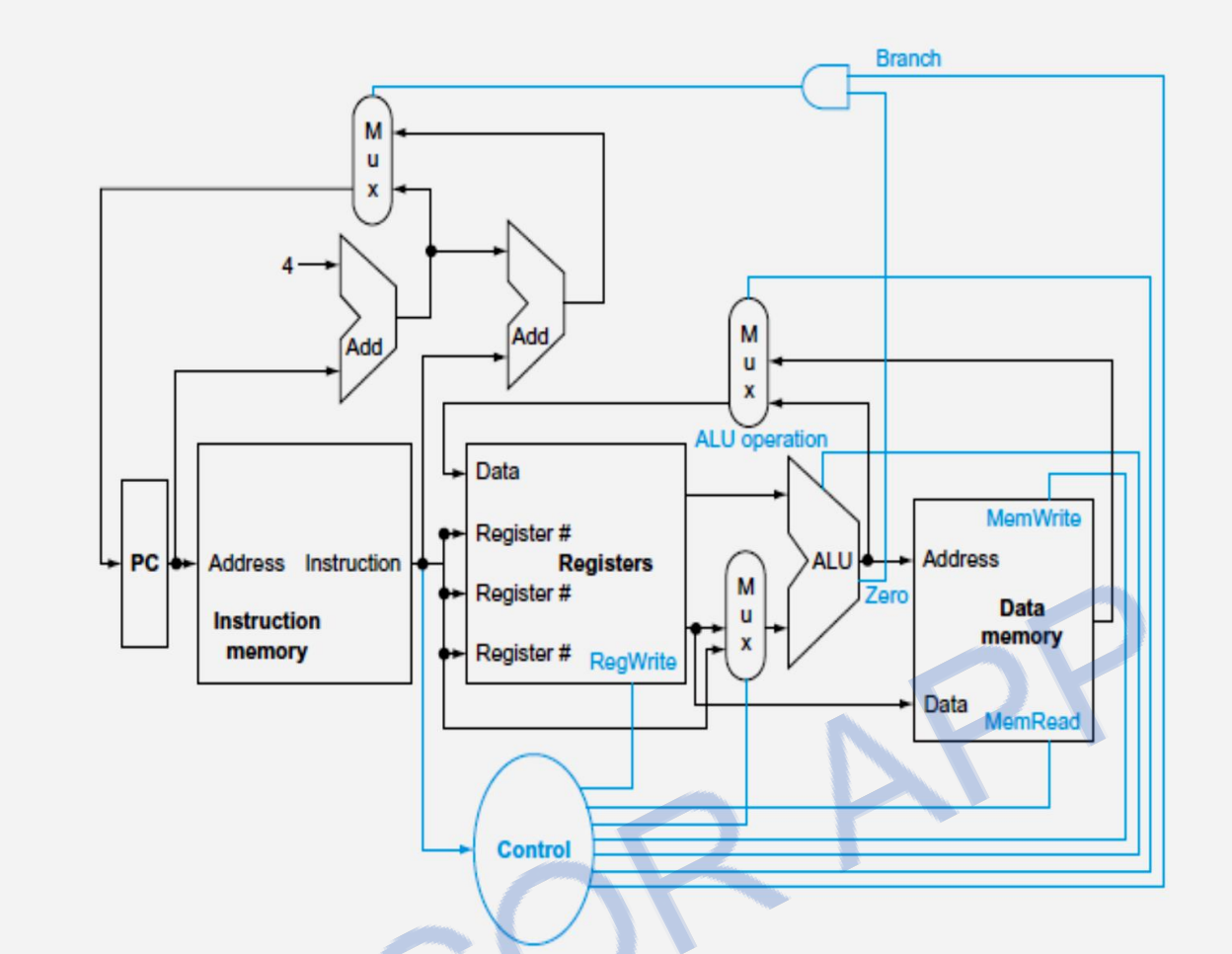


Fig 3.2: The basic implementation of the MIPS subset, including the necessary multiplexors and control lines.

- ❖ The top multiplexor controls the value of PC(PC+4 or the branch destination address.
- ❖ The middle multiplexor is used to steer the output of the ALU for writing in to the register file.
- ❖ The bottom most multiplexor is used to determine whether the second ALU input is from the registers

BUILDING DATAPATH

- **Single-cycle Datapath:**

  Each instruction executes in a single cycle

- **Multi-cycle Datapath:**

  Each instruction is broken up into a series of shorter steps

- **Pipelined Datapath:**

  Each instruction is broken up into a series of steps; Multiple instructions execute at once

Differences between single cycle and multi cycle datapath

❖ ***Single cycle Data Path*:**

- Each instruction is processed in one (long) clock cycle
- Two separate memory units for instructions and data.

❖ ***Multi-cycle Data Path*:**

- Divide the processing of each instruction into 5 stages and allocate one clock cycle per stage
- Single memory unit for both instructions and data
- Single ALU for all arithmetic operations
- Extra registers needed to hold values between each steps
  - Instruction Register (IR) holds the instruction
  - Memory Data Register (MDR) holds the data coming from memory
  - A, B hold operand data coming from the registers
  - ALUOut holds output coming out of the ALU

Creating a single cycle datapath

❖ This simplest datapath will attempt to execute all instructions in **one clock cycle**. This means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for instructions separate from one for data. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

❖ To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.

❖ A reasonable way to start a datapath design is to examine the major components required to execute each class of MIPS instructions. Let's start at the top by looking at which **datapath elements** each instruction needs, and then work our way down through the levels of **abstraction**. When we show the datapath elements, we will also

show their control signals. We use abstraction in this explanation, starting from the bottom up.

Datapath Element

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.

Program Counter (PC)

- Figure 3.3a shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address. Figure 3.3 b also shows the **program counter (PC)**, the register containing the address of the instruction in the program being executed.
- Lastly, we will need an adder to increment the PC to the address of the next instruction. This adder, which is combinational, can be built from the ALU simply by wiring the control lines so that the control always specifies an add operation.
- We will draw such an ALU with the label *Add*, as in Figure 3.3c, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.
- 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.

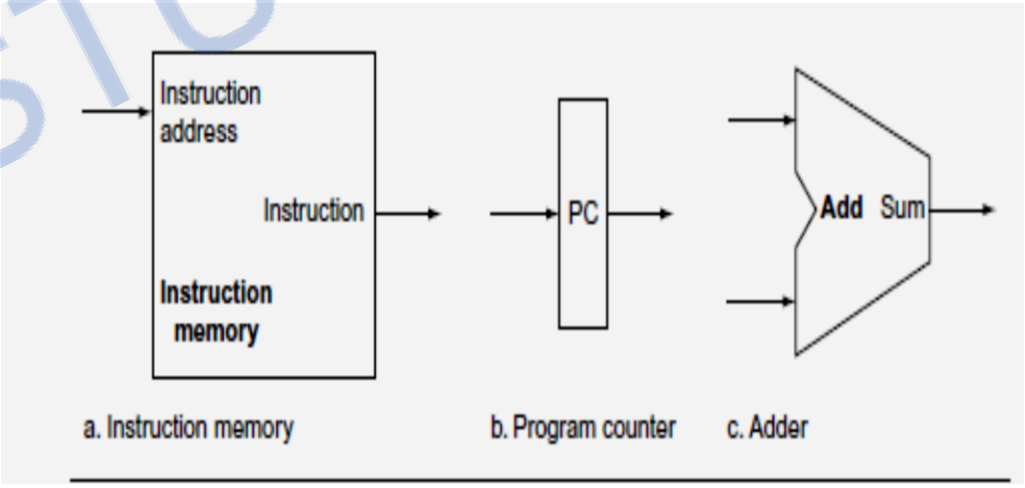


Fig 3.3: Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

Figure 3.4 shows how to combine the three elements to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

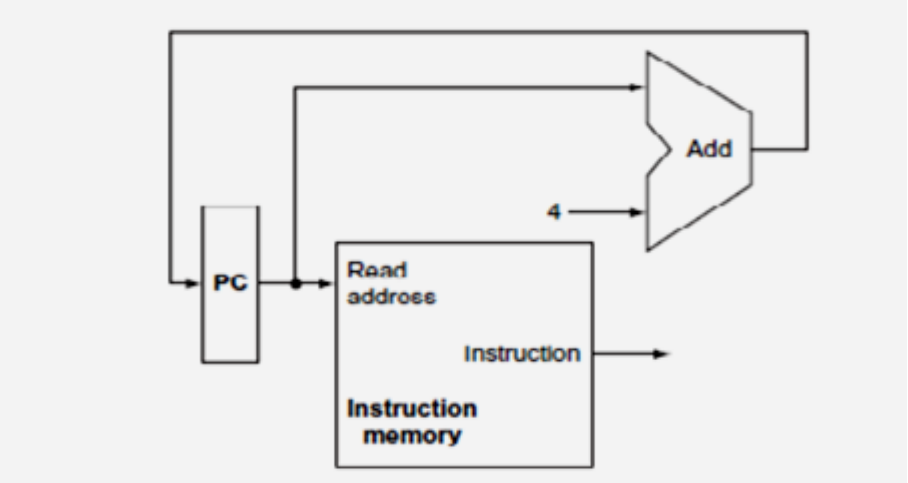


Fig 3.4: A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath

R-FORMAT INSTRUCTIONS

- To perform any operation we required two registers, perform an ALU operation on the contents of the registers, and write the result to a register. We call these instructions either *R-type instructions* or *arithmetic-logical instructions* (since they perform arithmetic or logical operations). This instruction class includes add, sub, AND, OR, and slt,
- 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.
- R-format instructions have three register operands, so we will need to read two data words from the register file and write one data word into the register file for each instruction.
- 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.
- 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 register file always outputs the contents of whatever register numbers are on the Read register inputs. Writes, however, are controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

- Figure 3.5a shows the result; 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.
- Figure 3.5b shows the 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.

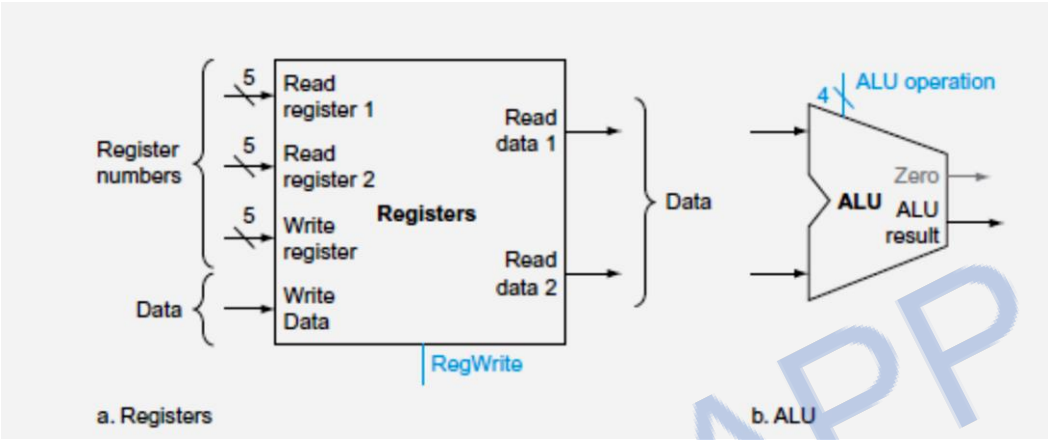


Fig 3.5: The two elements needed to implement R-format ALU operations are the register file and the ALU.

DATAPATH SEGMENT FOR *Load Word* and *Store Word* INSTRUCTION

- Now, consider the MIPS load word and store word instructions, which have the general form lw $t1,offset_value($t2) or sw $t1,offset_value ($t2).
- In these instructions $t1 is a data register and $t2 is a base register. The memory address is computed by adding the base register($t2), to the 16-bit signed off set, values specified in the instruction.
- If the instruction is a store, the value to be stored must also be read from the register file where it resides in $t1. If the instruction is a load, the value read from memory must be written into the register file in the specified register, which is $t1. Thus, we will need both the register file and the ALU from Figure 3.5.
- In addition, we will need a unit to sign-extend the 16-bit off set field in the instruction to a 32-bit signed value, and a data memory unit to read from or write to. The data memory must be written on store instructions; Figure 3.6 shows these two elements.

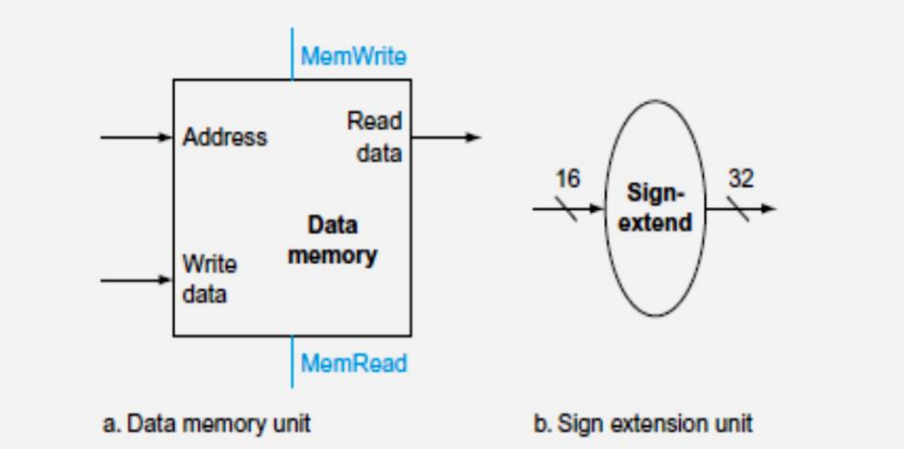


Fig 3.6: The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 3.5

DATAPATH SEGMENT FOR *Branch* INSTRUCTION

For computing the branch target address, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction at the branch target address.

When the condition is true (i.e., the operands are equal), the branch target address becomes the new PC, and we say that the **branch is taken**. If the operands are not equal, the incremented PC should replace the current PC (just as for any other normal instruction); in this case, we say that the **branch is not taken**.

Figure 3.7 shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes a sign extension unit, and an adder.

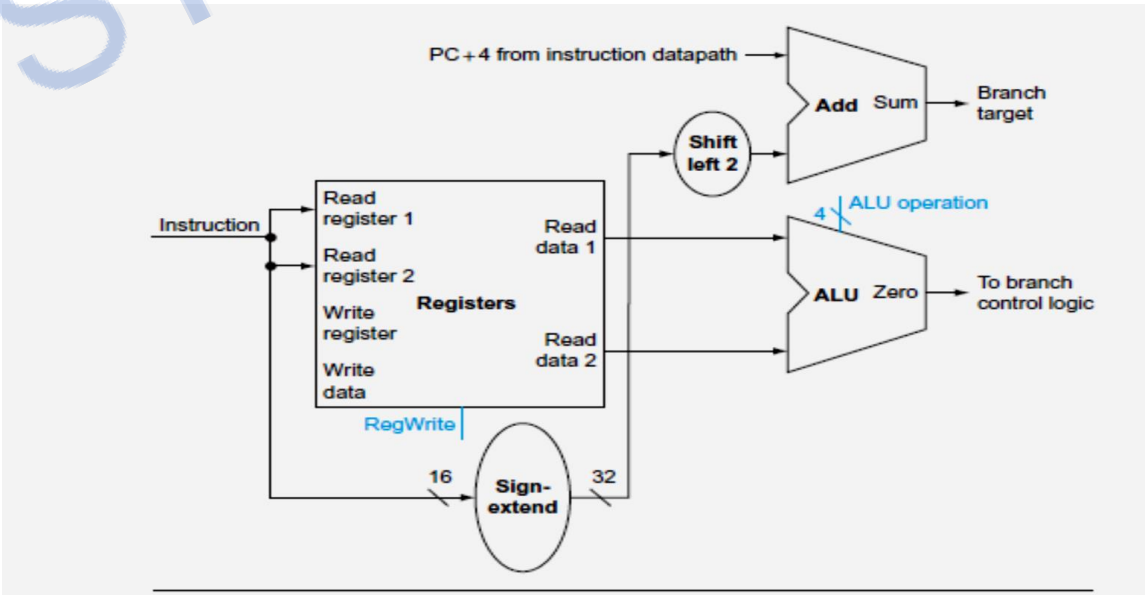


FIG 3.7: Computation of branch target address

Since that ALU provides an output signal that indicates whether the result was 0, we can send the two register operands to the ALU with the control set to do a subtract. If the Zero signal out of the ALU unit is asserted, we know that the two values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equal test of branches. Later, we will show exactly how to connect the control signals of the ALU for use in the datapath.

The jump instruction operates by replacing the lower 28 bits of the PC with the lower 26 bits of the instruction shifted left by 2 bits.

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.

The 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

Table 3.1: ALU control signals

Depending on the instruction class, the ALU will need to perform one of these first five functions. 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, which we call ALUOp.

ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract (01) for beq, or determined by the operation encoded in the funct field (10). 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 shown previously. In Figure 3.2, we show how to set the ALU control inputs based on the 2-bit ALUOp control and the 6-bit function code. Later in this chapter we will see how the ALUOp bits are generated from the main control unit.

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

Table 3.2: The ALUOp control bits and the different function codes for the R-type instruction.

Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially increase the speed of the control unit. There are several different ways to implement the mapping from the 2-bit ALUOp field and the 6-bit funct field to the four ALU operation control bits.

Table 3.3; shows the truth table how the 4-bit ALU control is set depending on these two input fields. Since the full truth table is very large (28 = 256 entries), we show only the truth table entries for which the ALU control must have a specific value

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Table 3.3: The truth table for the 4 ALU control bits (called Operation)

Designing the Main Control Unit

To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the three instruction classes: the R-type, branch, and load-store instructions. Figure 3.8 shows these formats.

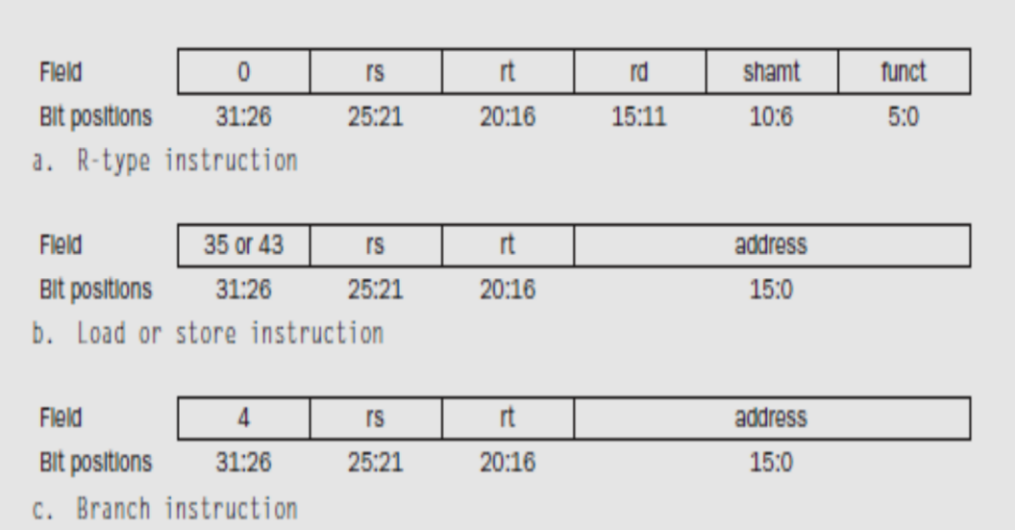


FIGURE 3.8 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

There are several major observations about this instruction format that we will rely on:

■ The op field, is called the opcode, is always contained in bits 31:26. Refer to this field as Op[5:0].

■ 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).

Th us, we will need to add a multiplexor to select which fi eld of the instruction is used to indicate the register number to be written. The first design principle simplicity favors regularity−pays off here in specifying control.

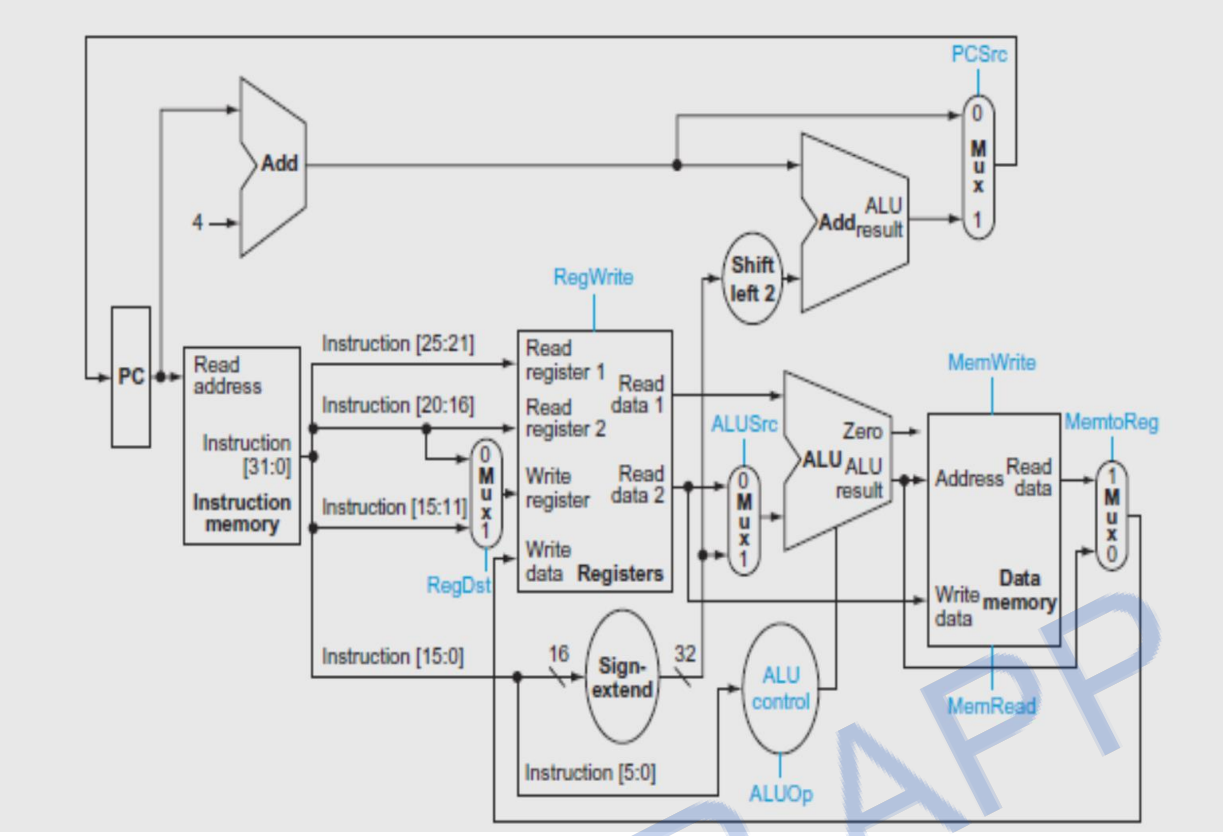


FIGURE 3.9 The datapath with all necessary multiplexors and all control lines identified.

- Figure 3.9 shows seven single-bit control lines plus the 2-bit ALUOp control signal. We have already defined how the ALUOp control signal works, and it is useful to define what the seven other control signals do informally before we determine how to set these control signals during instruction execution.
- Table 3.4 describes the function of these seven control lines. That control line should be asserted (activated or set true) if the instruction is branch on equal (a decision that the control unit can make) and the Zero output of the ALU, which is used for equality comparison, is asserted. To generate the PCSrc signal, we will need to AND together a signal from the control unit, which we call Branch, with the Zero signal out of the ALU.

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.

Table 3.4: The effect of each of the seven control signals.

- These nine control signals (seven from Figure 4.16 and two for ALUOp) can now be set on the basis of six input signals to the control unit, which are the opcode bits 31 to 26. Figure 3.10 shows the datapath with the control unit and the control signals.

Finalizing Control

Table 3.6 shows the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function,

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Table 3.6: The control function for the simple single-cycle implementation is completely specified by this truth table.

OPERATION OF THE DATAPATH

The flow of three different instruction classes through the datapath is shown in table 3.5. The asserted control signals and active datapath elements are highlighted in each of these. Note

that a multiplexor whose control is 0 has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Table 3.5: The setting of the control lines is completely determined by the opcode fields of the instruction.

Fig 3.10: shows the datapath with the control unit and the control signals. The setting of the control lines depends only on the opcode, we defi ne whether each control signal should be 0, 1, or don't care (X) for each of the opcode values.

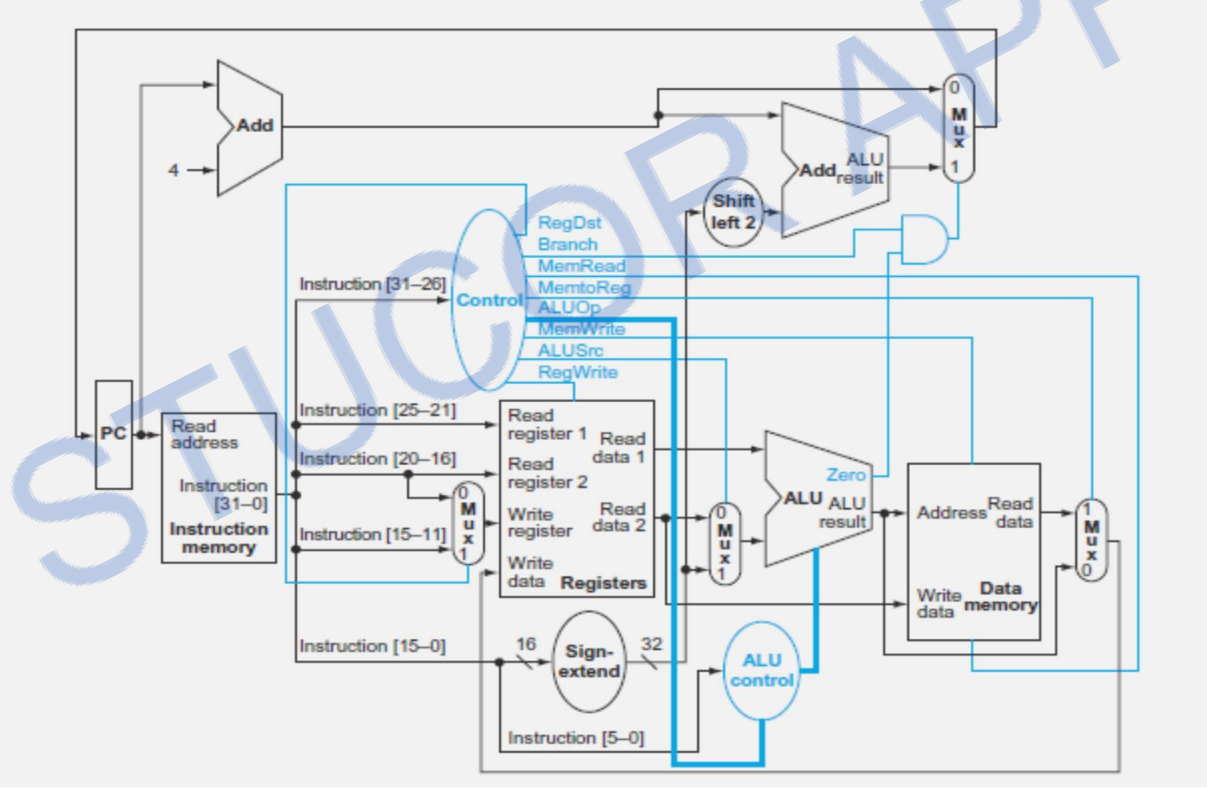


FIGURE 3.10: The simple datapath with the control unit.

DATAPATH FOR THE OPERATION OF A *R-TYPE* INSTRUCTION

Figure 3.11 shows the operation of the datapath for an R-type instruction, such as add $t1,$t2,$t3. Although everything occurs in one clock cycle, we can think of four steps to execute the instruction; these steps are ordered by the flow of information:

1. The instruction is fetched, and the PC is incremented.

2. Two registers, $t2 and $t3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.

3. The ALU operates on the data read from the register file, using the function code (bits 5:0, which is the funct field, of the instruction) to generate the ALU function.

4. The result from the ALU is written into the register file using bits 15:11 of the instruction to select the destination register ($t1).

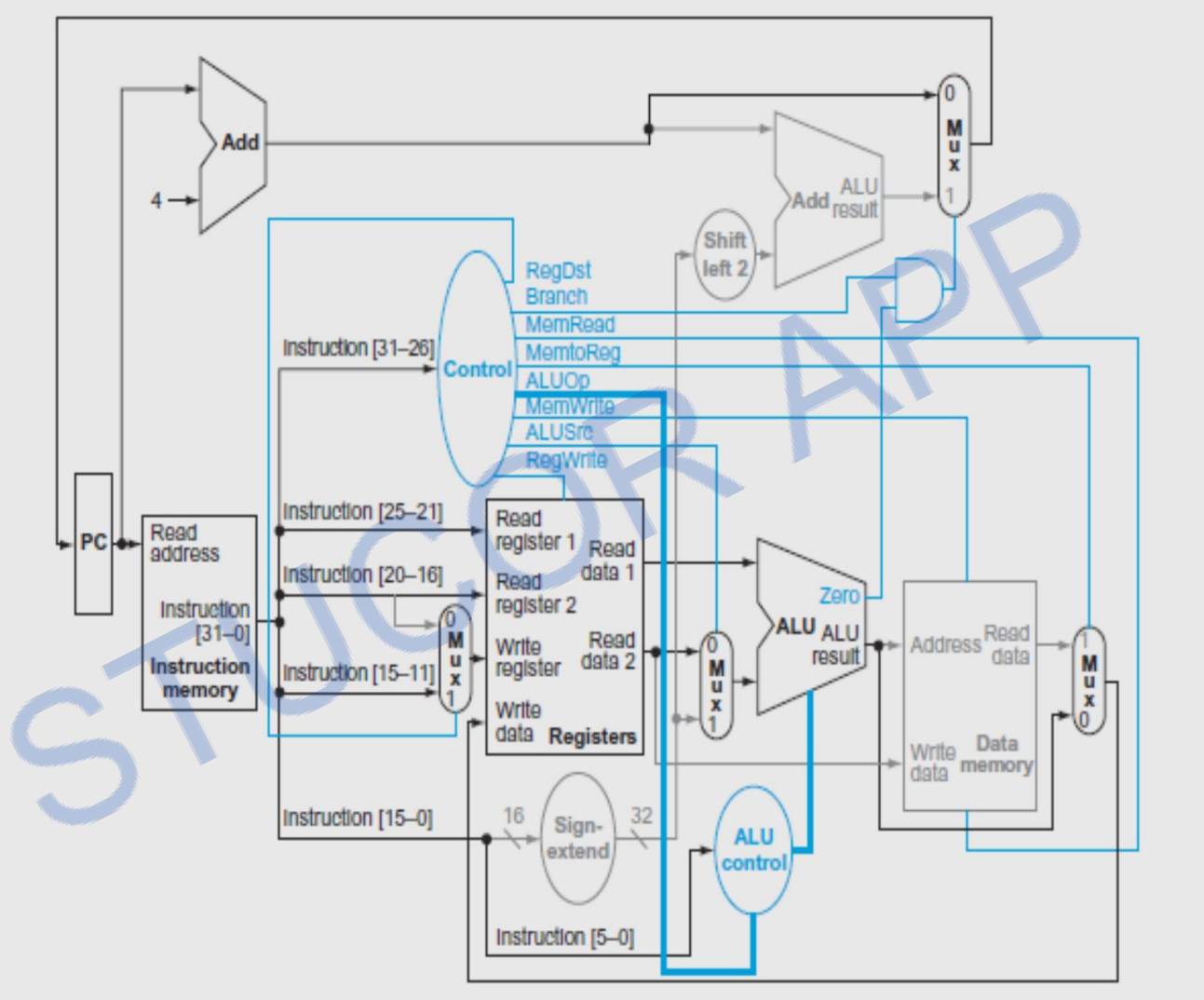


FIGURE 3.11: The datapath in operation for an R-type instruction

DATAPATH FOR THE OPERATION OF *load word* INSTRUCTION

The given figure 3.12 shows the active functional units and asserted control lines for a load. We can think of a load instruction as operating in five steps (similar to how the R-type executed in four):

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. A register ($t2) value is read from the register file.

3. The ALU computes the sum of the value read from the register file and the sign-extended, lower 16 bits of the instruction (offset).

4. The sum from the ALU is used as the address for the data memory.

5. The data from the memory unit is written into the register file; the register destination is given by bits 20:16 of the instruction ($t1).

Finally, we can show the operation of the branch-on-equal instruction, such as beq $t1, $t2, offset, in the same fashion. It operates much like an R-format instruction, but the ALU output is used to determine whether the PC is written with PC + 4 or the branch target address.

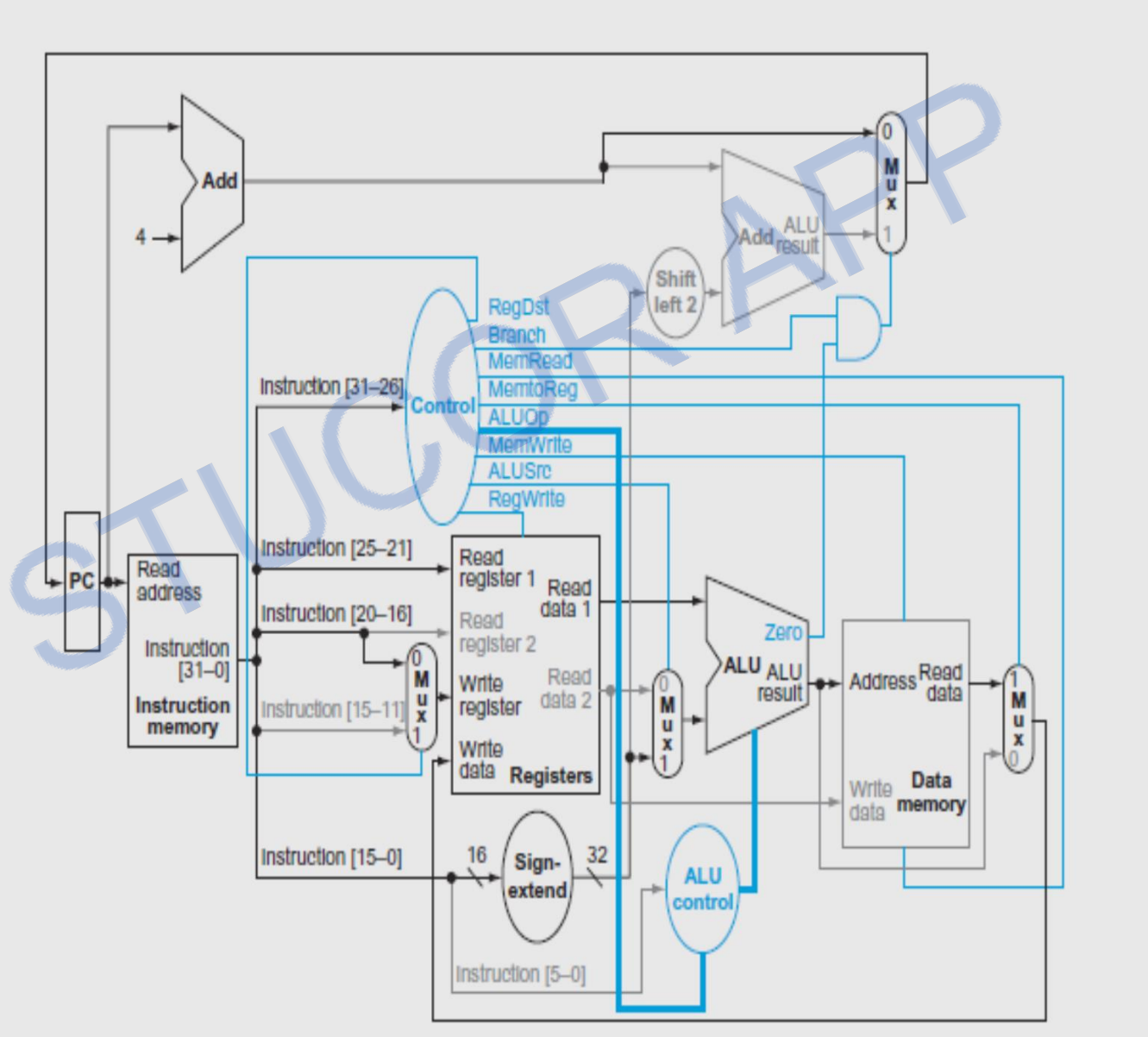


FIGURE 3.12: The datapath in operation for a load instruction.

DATAPATH FOR THE OPERATION OF *BRANCH-ON-EQUAL* INSTRUCTION

The given figure 3.13 shows the four steps in execution:

1. An instruction is fetched from the instruction memory, and the PC is incremented.

2. Two registers, $t1 and $t2, are read from the register file.

3. The ALU performs a subtract on the data values read from the register file. The value of PC + 4 is added to the sign-extended, lower 16 bits of the instruction (offset) shifted left by two; the result is the branch target address.

4. The Zero result from the ALU is used to decide which adder result to store into the PC.

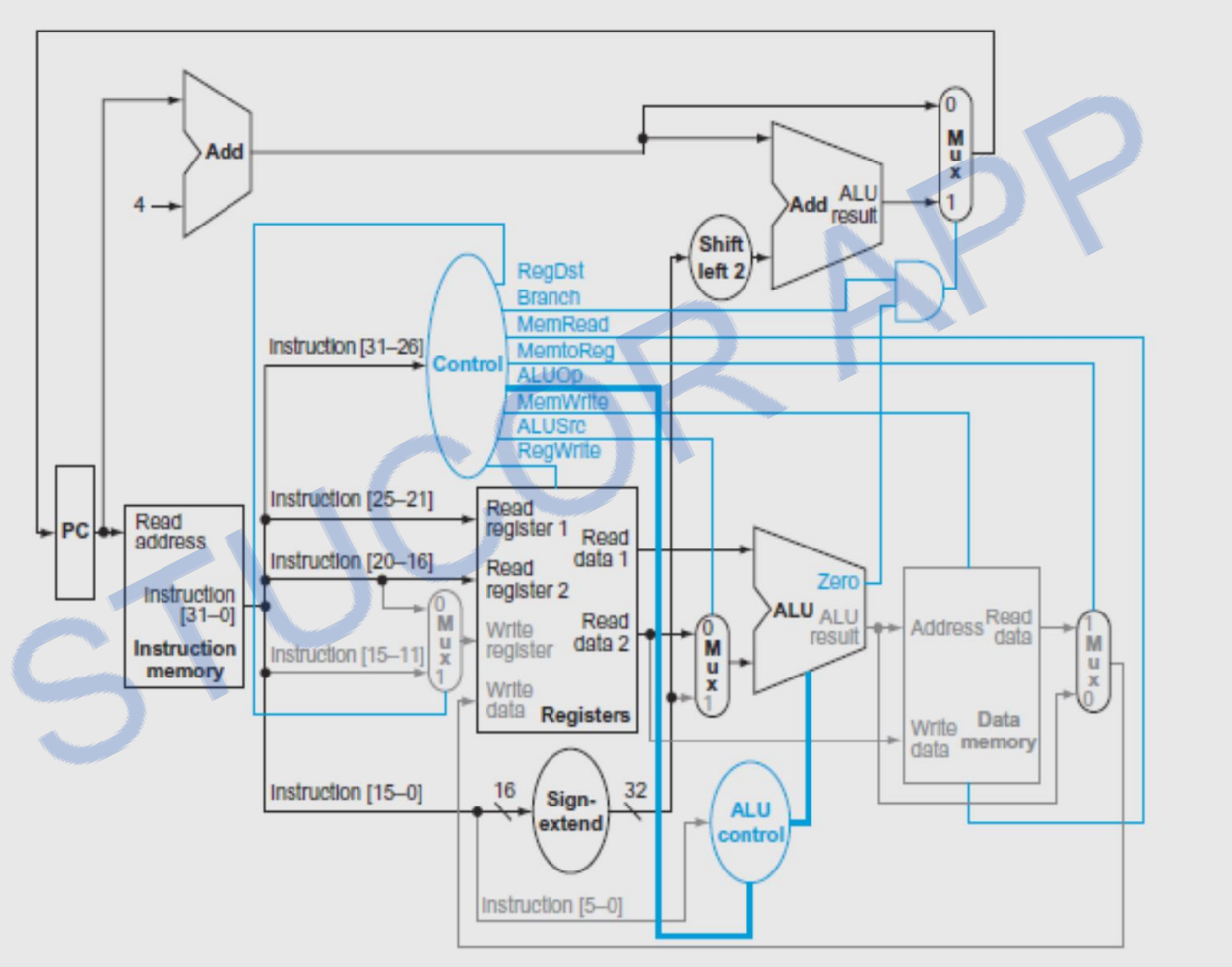


FIGURE 3.13: The datapath in operation for a branch-on-equal instruction.

PIPELINING

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an

instruction in parallel. The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

Today,. The *non-pipelined* approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, start over with the next dirty load.

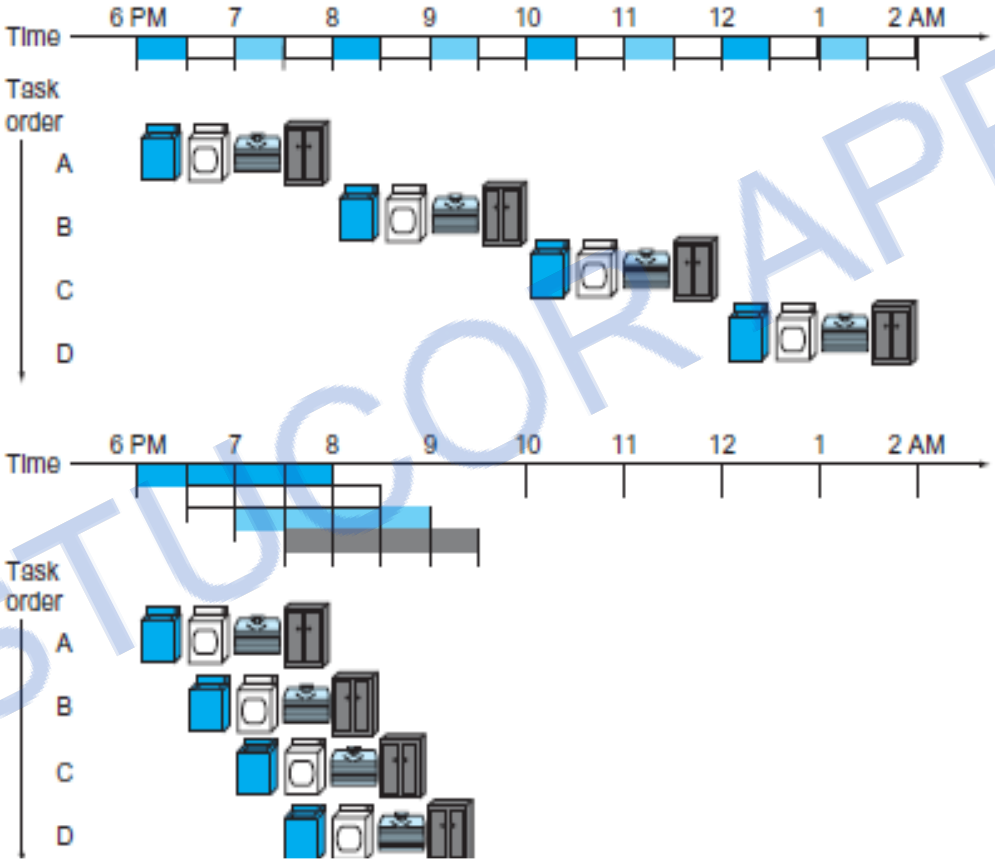


FIGURE 3.14: The laundry analogy for pipelining.

- A technique used in advanced microprocessors where the microprocessor begins executing a second instruction before the first has been completed.
- A Pipeline is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages.

- With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing arithmetic operations, holding them in a buffer close to the processor until each instruction operation can performed.

How Pipelines Works

- The pipeline is divided into segments and each segment can execute it operation concurrently with the other segments. Once a segment completes an operations, it passes the result to the next segment in the pipeline and fetches the next operations from the preceding segment.

- The *pipelined* approach takes much less time, as Figure 3.14 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.

- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline, in this case four: washing, drying, folding, and putting away. Therefore, pipelined laundry is potentially four times faster than non-pipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. It's only 2.3 times faster in 3.14, because we only show 4 loads.

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.

DESIGNING INSTRUCTION SETS FOR PIPELINING

- First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
- Second, MIPS have only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.
- Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.
- Fourth, operands must be aligned in memory. Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

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.

Hazards

- Structural Hazards
- Data Hazards
- Control Hazards

STRUCTURAL HAZARD

- Structural Hazard occurs when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
- A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

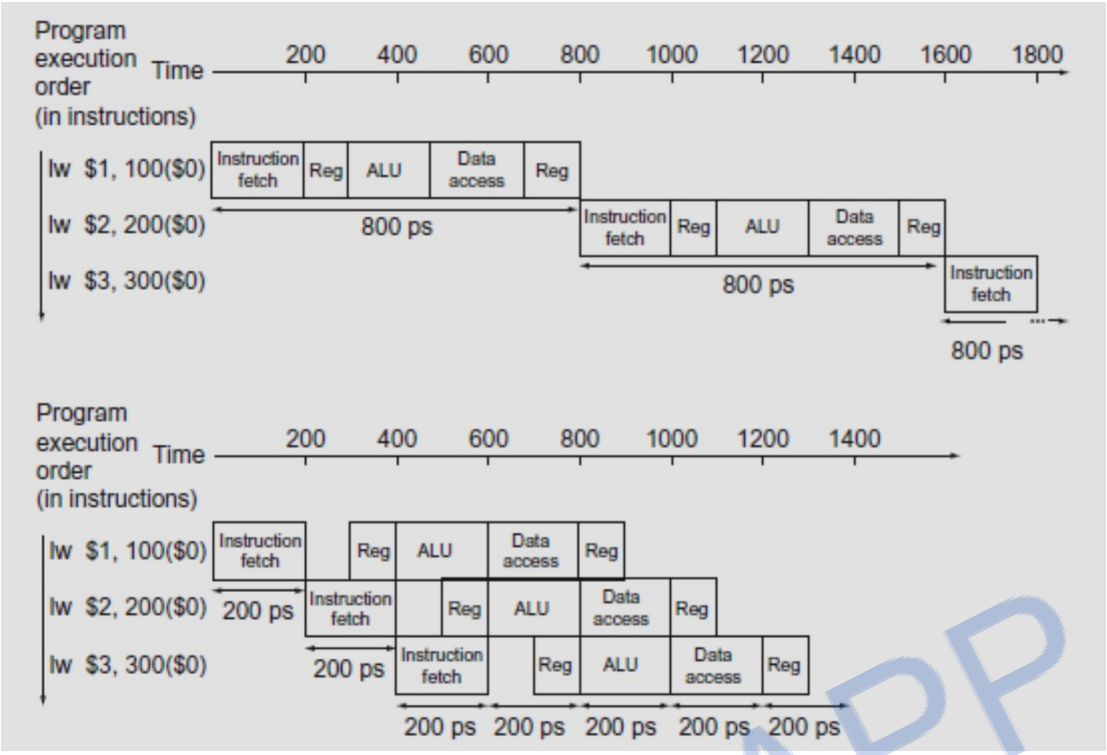


FIGURE 3.15 Single-cycle, non-pipelined execution in top versus pipelined execution in bottom.

As we said above, 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 3.15 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 HAZARDS

- It is also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.
- **Data hazards** occur when the pipeline must be stalled because one step must wait for another to complete.
- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. 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

- Without intervention, 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.
- To resolve the data hazard, for the code sequence above, as soon as the ALU creates the sum for the add operation, we can supply it as an input for the subtract. This is done by adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding or bypassing.**
- Figure below shows the connection to forward the value in $s0 after the execution stage of the add instruction as input to the execution stage of the sub instruction.

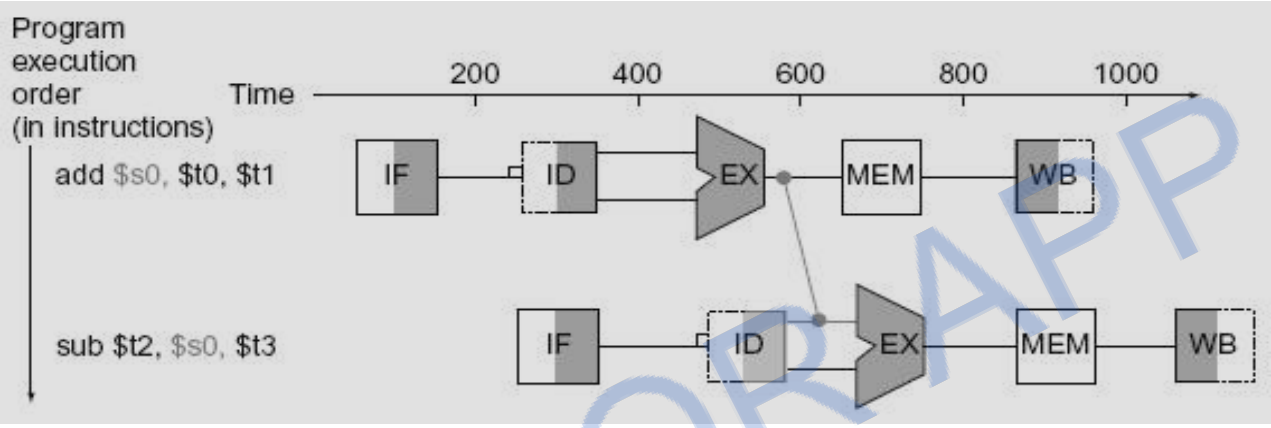
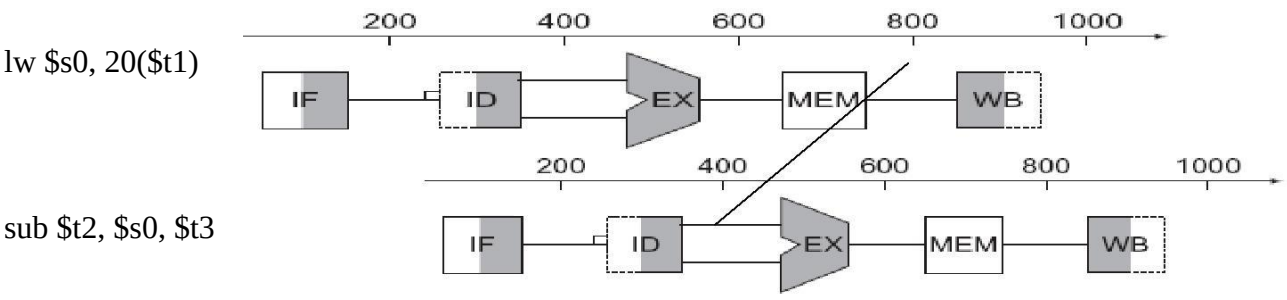


Fig 3.16: **Graphical representation of forwarding**

- Forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.
- Forwarding cannot prevent all pipeline stalls, suppose the first instruction was a load of $s0 instead of an add, So 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 instruction.



- Even with forwarding, we would have to stall one stage for a load-use data hazard, this figure shows below an important pipeline concept, officially called a pipeline stall, but

often given the nickname bubble.

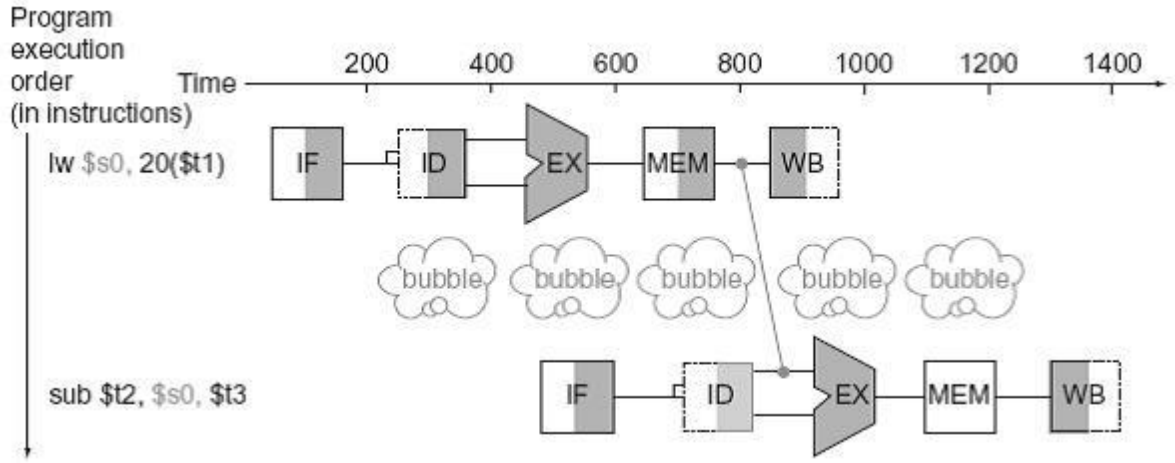


Fig 3.17: A stall even with forwarding when an R-format instruction following a load tries to use the data.

CONTROL HAZARDS

- It is also called as 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.
- A control hazard, arising from the need to make a decision based on the results of one instruction while others are executing.
- Even with this extra hardware, the pipeline involving conditional branches would look like figure 3.18. The **lw** instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before starting.
- The equivalent decision task in a computer is the branch instruction. Notice that we must begin fetching the instruction following the branch on the very next clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory.
- One possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from. Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline.

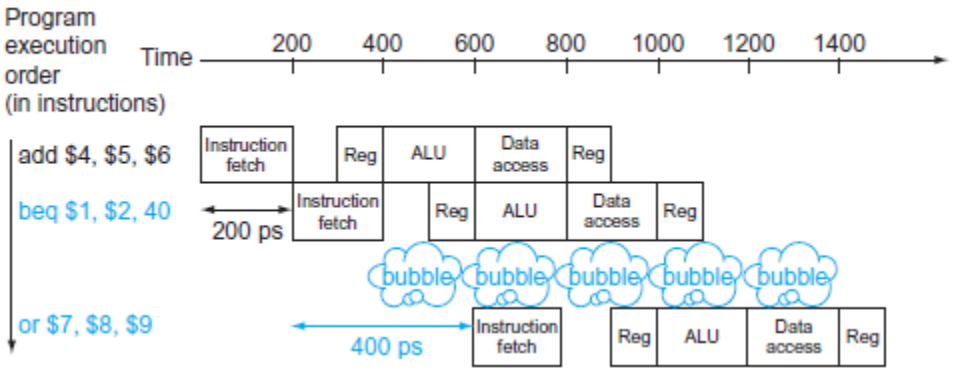


FIGURE 3.18: Pipeline showing stalling on every conditional branch as solution to control hazards.

BRANCH PREDICTION

A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Static branch prediction

A more sophisticated version of **branch prediction** would have some branches predicted as taken and some as untaken. In the case of programming, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.

Dynamic branch prediction

Dynamic hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next **prediction** depending on the success of recent guesses. One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.

PIPELINED DATAPATH

Figure 3.19 shows the single-cycle datapath from 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 datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register fi le read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

❖ In Figure 3.19, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.

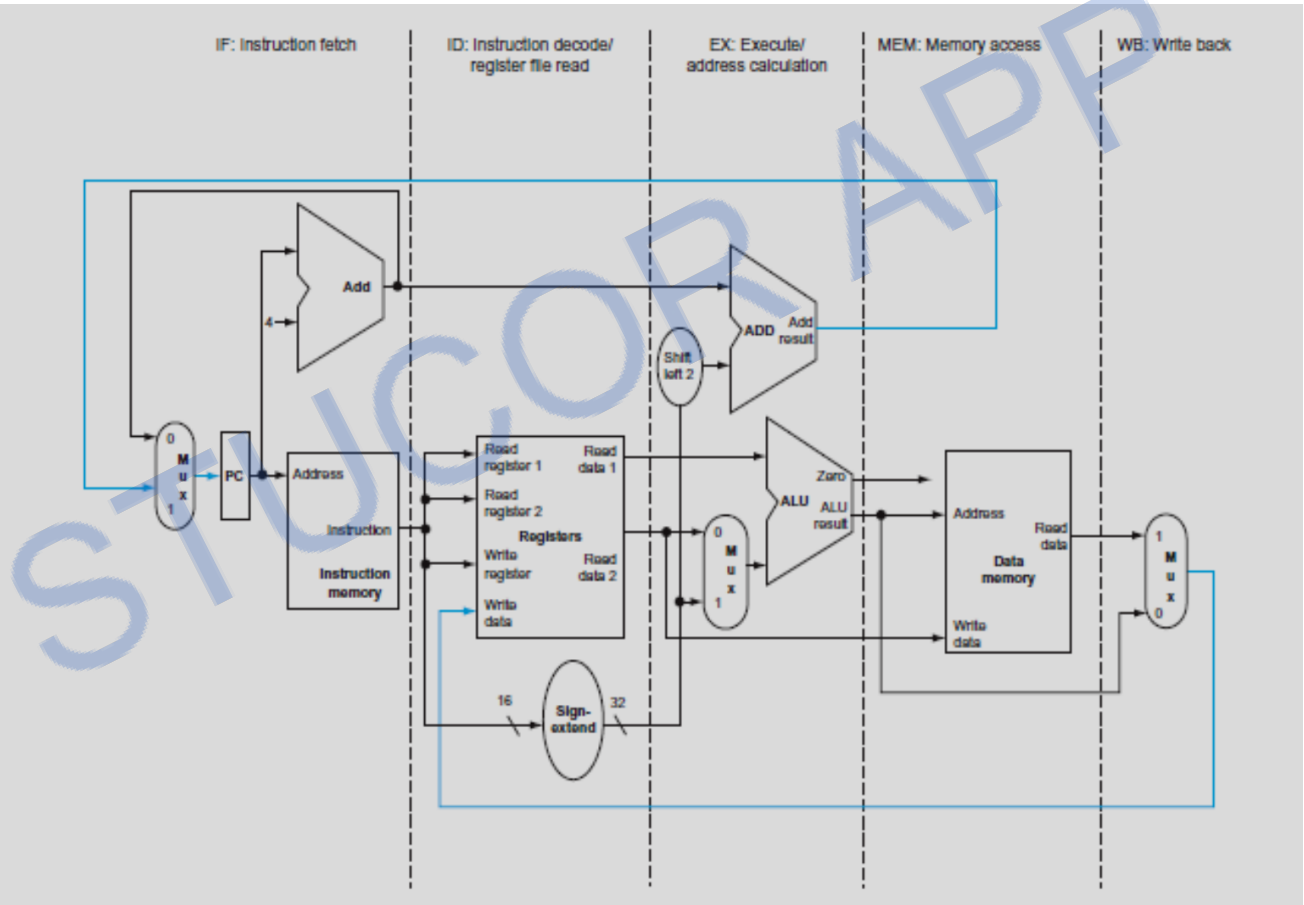


Fig 3.19: The single-cycle datapath

❖ Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward. 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 fi le in the middle of the datapath

■ The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

- Data flowing from right to left does not aff ect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that the fi rst right-to-left flow of data can lead to data hazards and the second leads to control hazards.
- One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship. Figure 3.20 shows the execution of the instructions in Figure 4.27 by displaying their private datapaths on a common timeline. Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.

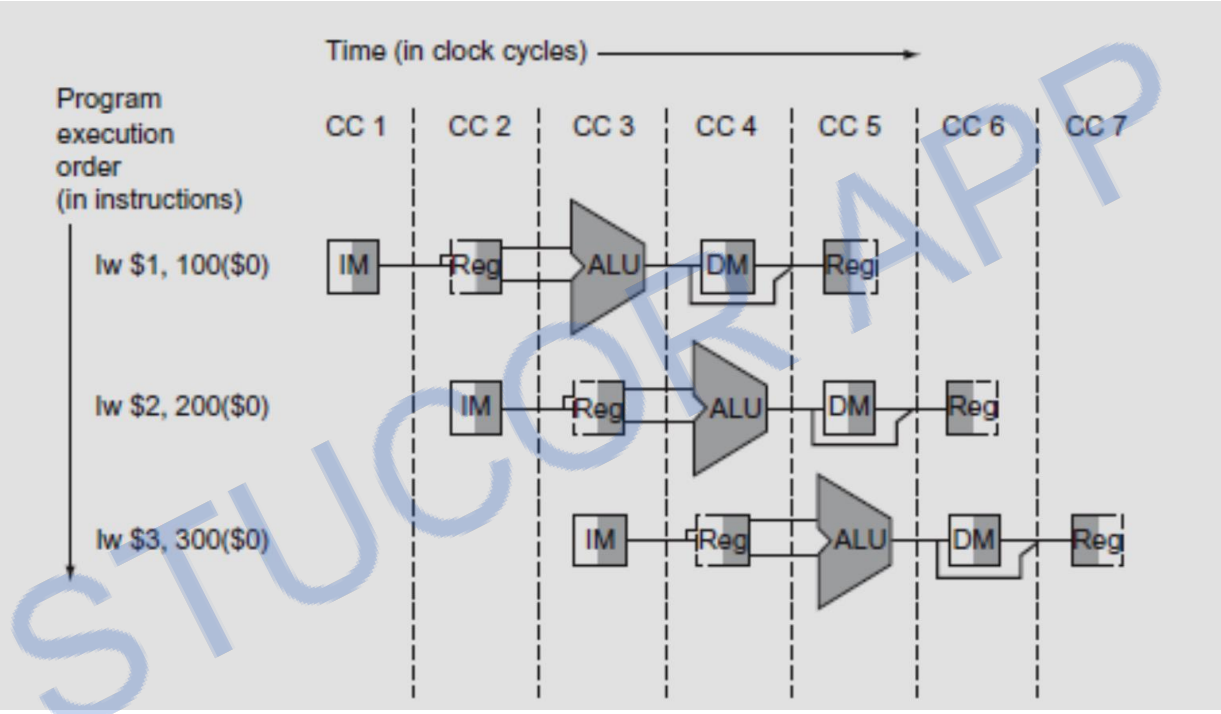


Fig 3.20: Instructions being executed using the single-cycle datapath in Figure 3.19, assuming pipelined execution.

- For example, as Figure 4.34 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages.
- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

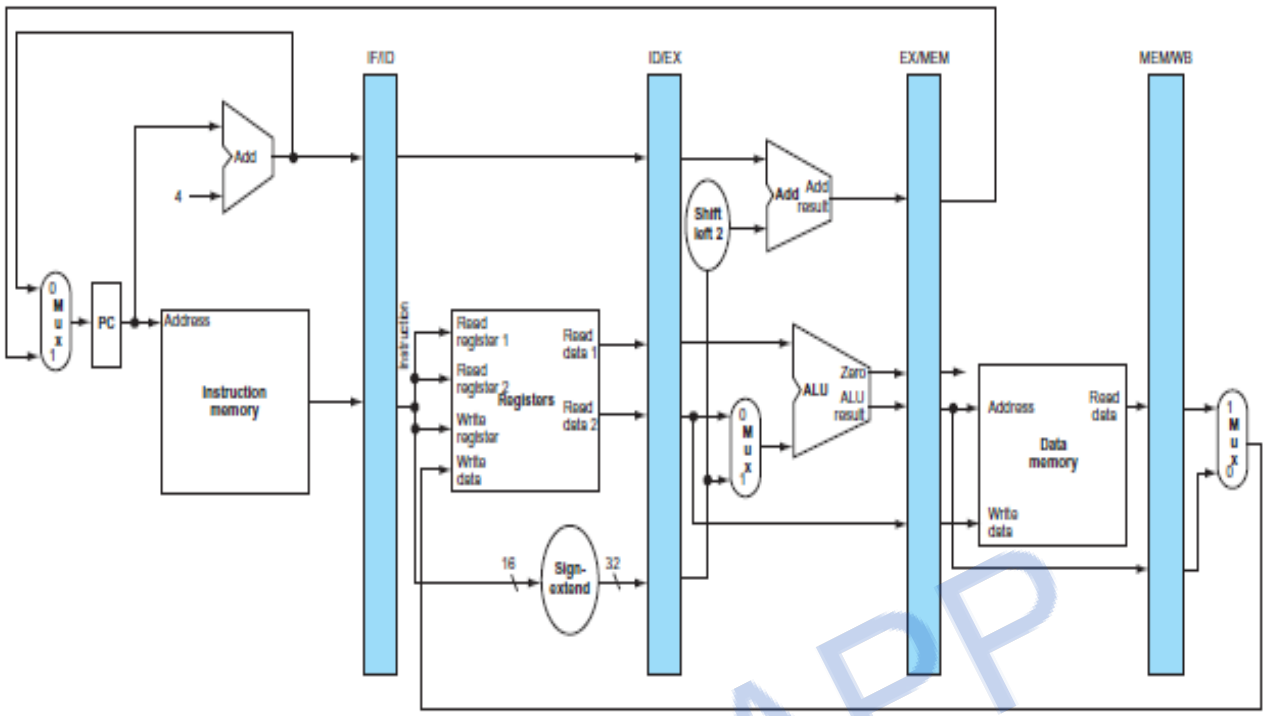


FIGURE 3.21: The pipelined version of the datapath

- Figure 3.21 shows the pipelined datapath 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. Notice that there is no pipeline register at the end of the write-back stage.
- All instructions must update some state in the processor—the register file, memory, or the PC.

EXECUTION OF *load* INSTRUCTION IN A PIPELINED DATAPATH

Figures 3.22 through 3.24, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. The five stages are the following:

1. *Instruction fetch:* The top portion of Figure 3.22 shows the instruction being 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.

2. Instruction decode and register file read: The bottom portion of Figure 3.22 shows the instruction portion of 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.

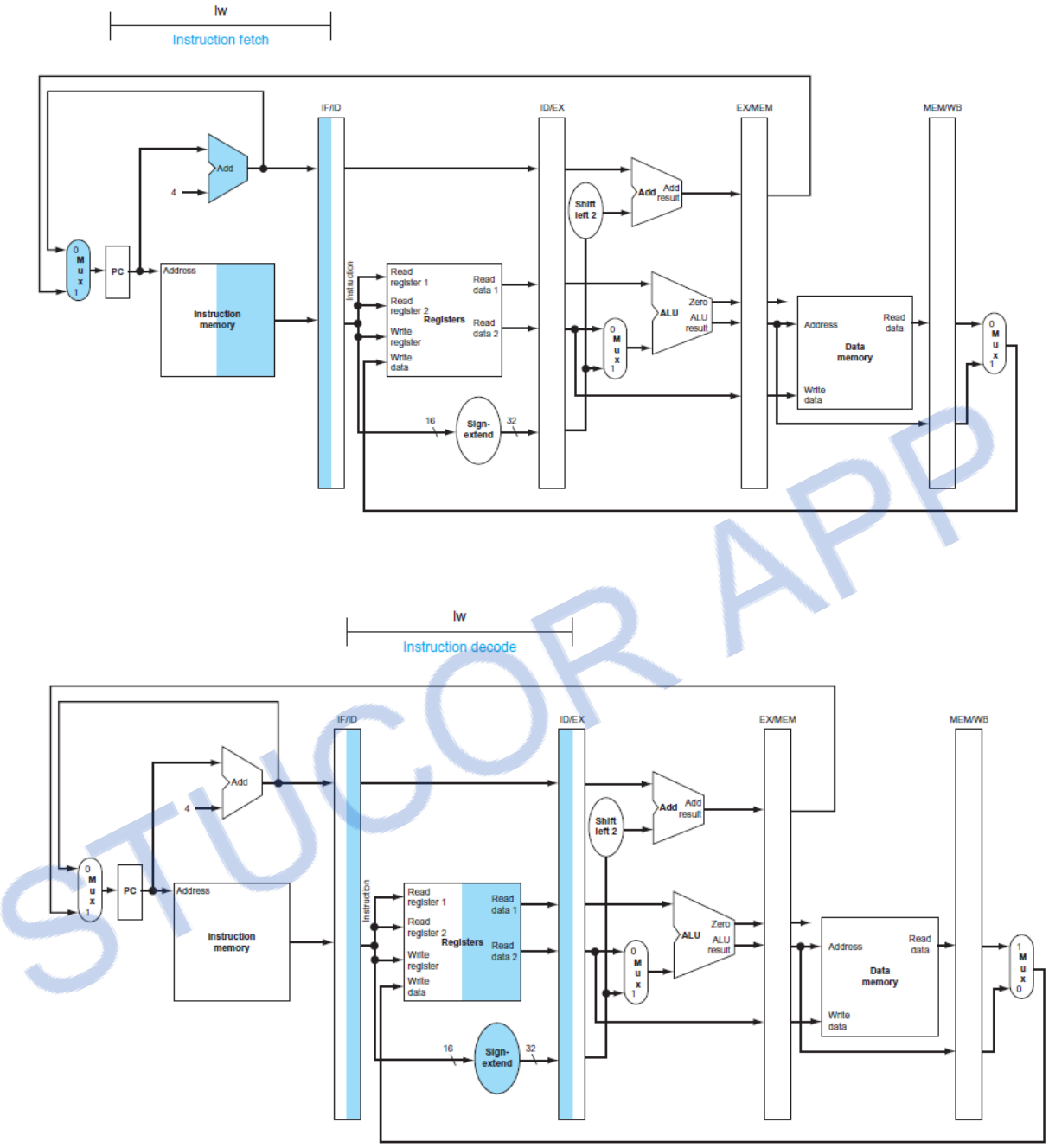


FIGURE 3.22: IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 3.21 highlighted.

3. Execute or address calculation: Figure 3.23 shows that 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.

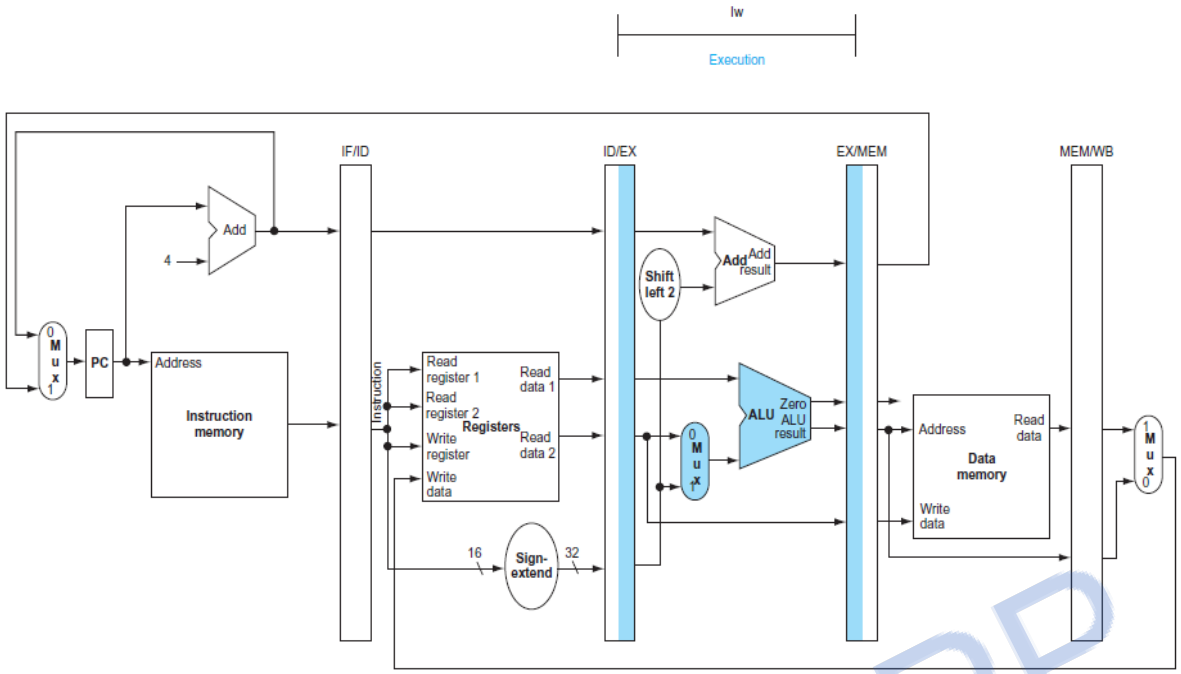


FIGURE 3.23 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 3.21 used in this pipe stage.

4. Memory access: The top portion of Figure 3.24 shows 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 bottom portion of Figure 3.24 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure. This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register.

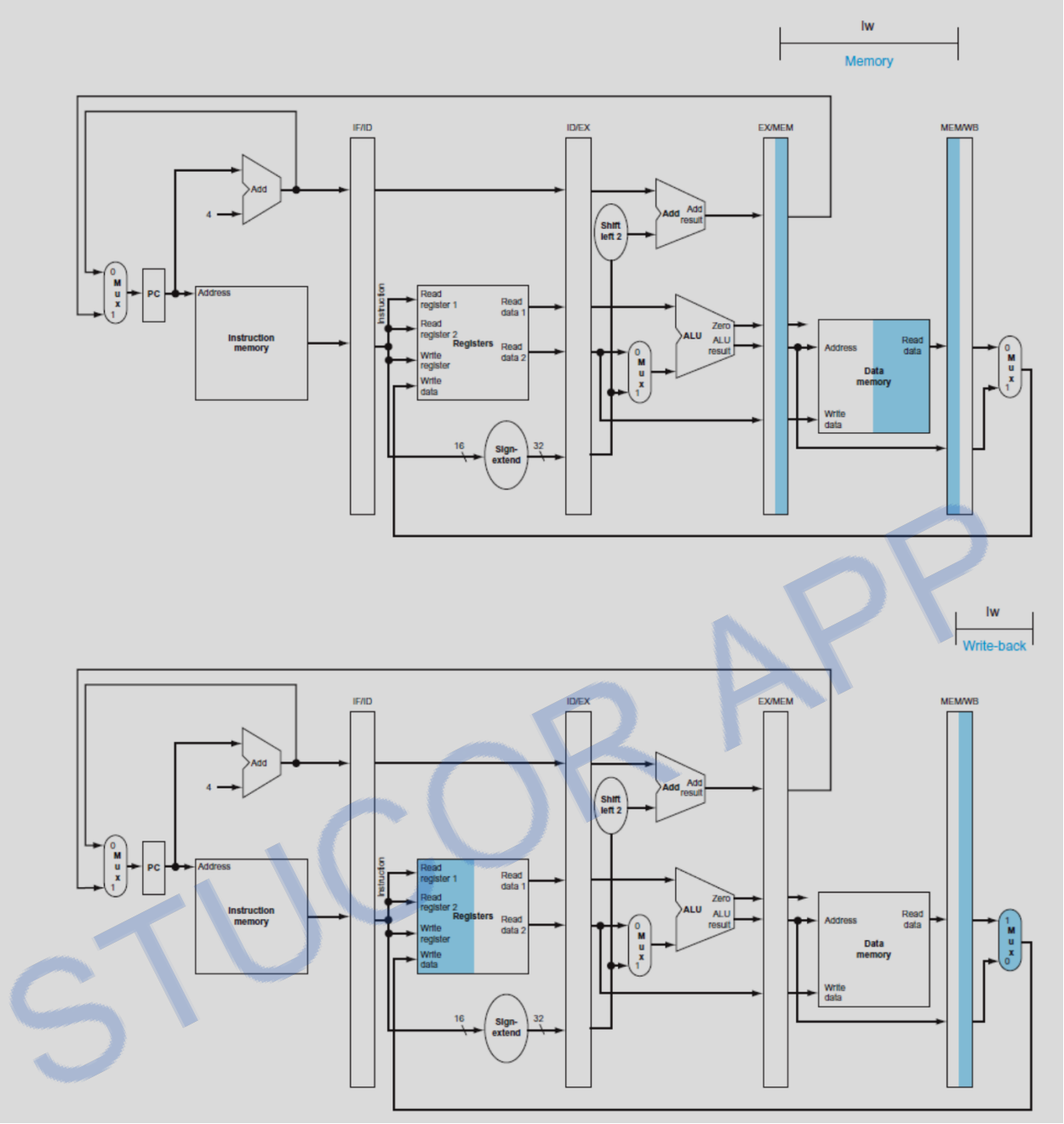


FIGURE 3.24 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 3.21 used in this pipe stage.

EXECUTION OF *Store* INSTRUCTION IN A PIPELINED DATAPATH

Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are the five pipe stages of the 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, so the top portion of Figure 3.25 works for store as well as load.

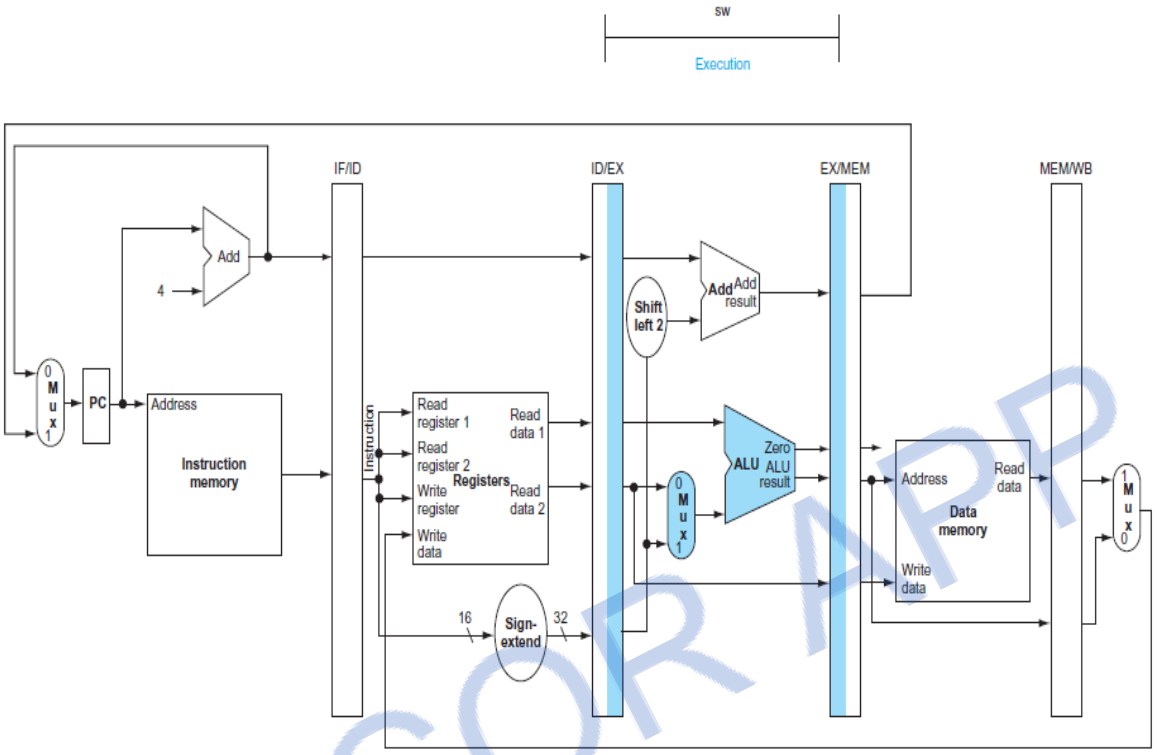


FIGURE 3.25 EX: The third pipe stage of a store instruction

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. The bottom portion of Figure 3.25 for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

3. Execute and address calculation: Figure 3.26 shows the third step; the effective address is placed in the EX/MEM pipeline register.

4. Memory access: The top portion of Figure 3.26 shows 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.

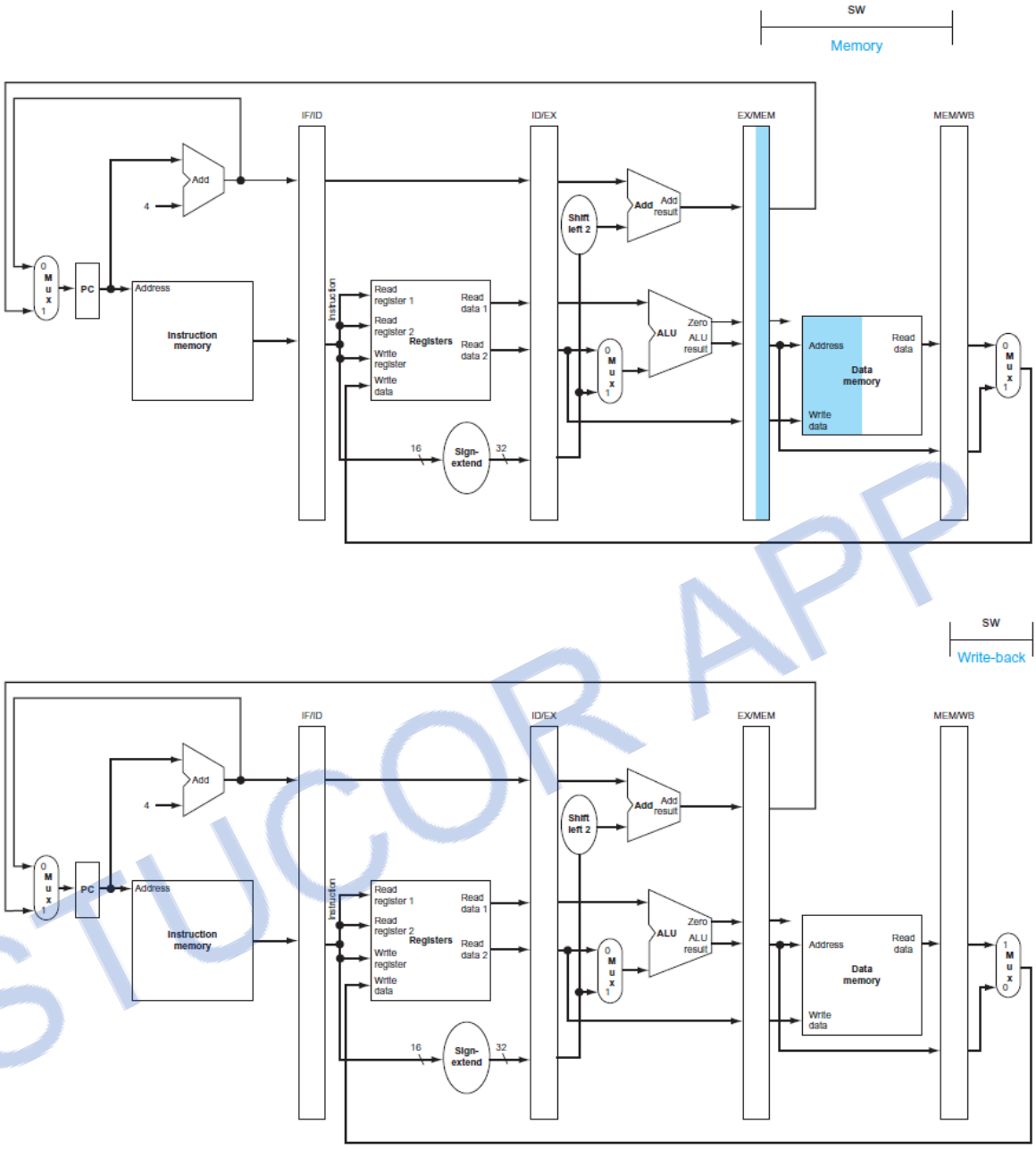


FIGURE 3.26: MEM and WB: The fourth and fifth pipe stages of a store instruction.

5. Write-back: The bottom portion of Figure 3.26 shows the final step of the store. For this instruction, nothing happens in the write-back stage.

- For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

PIPELINED CONTROL

Adding control to the pipelined datapath is referred to as pipelined control. It is started with a simple design that views the problem through pipeline bars in between the stages. The first step is to label the control lines on the existing datapath. Figure 3.27 shows those lines.

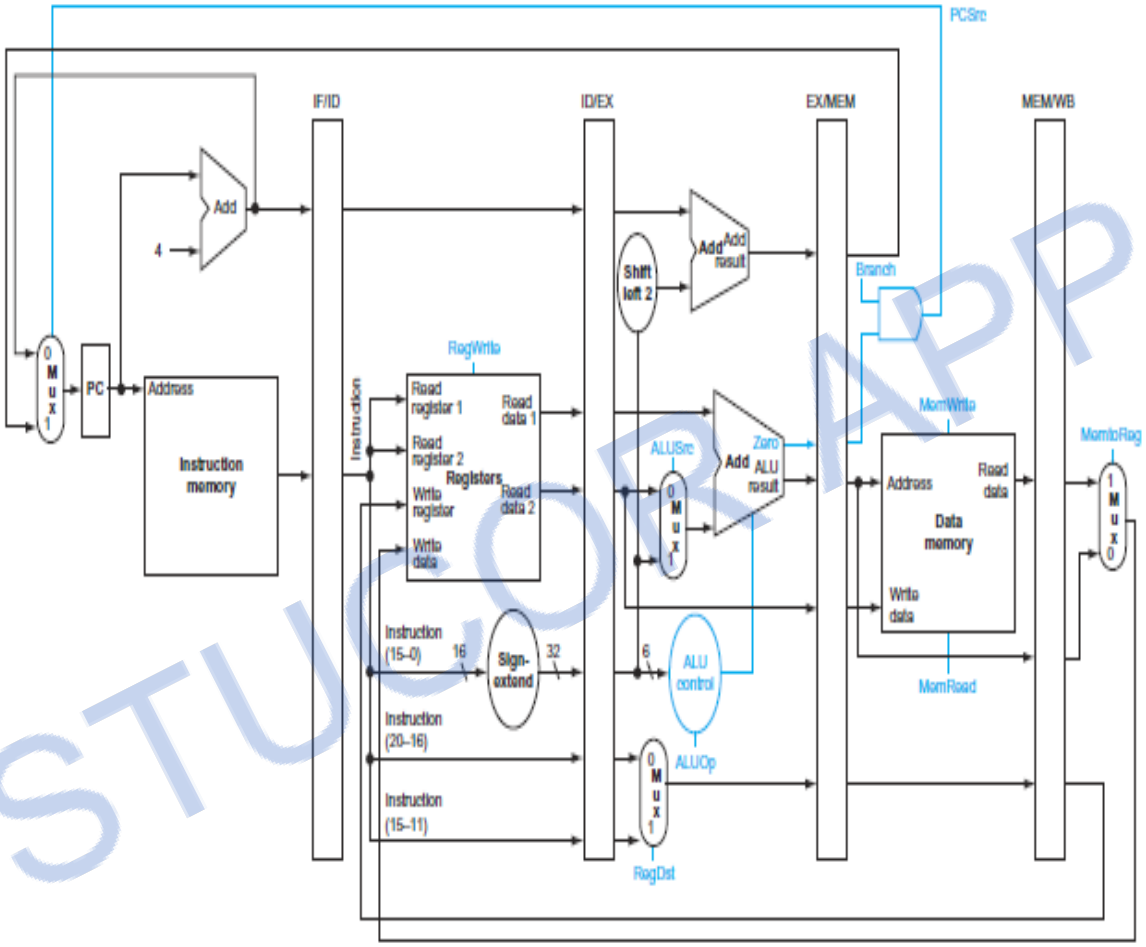


FIGURE 3.27: The pipelined datapath with the control signals identified.

- To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

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 (see Figures 4.48). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
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.

FIGURE 4.48 A copy of Figure 4.16. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.47. When a 1-bit control to a 2-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 4.46. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURE 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

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. Recall that PCSrc in Figure 4.48 selects the next sequential address unless control asserts Branch and the ALU result was 0.

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. Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values.

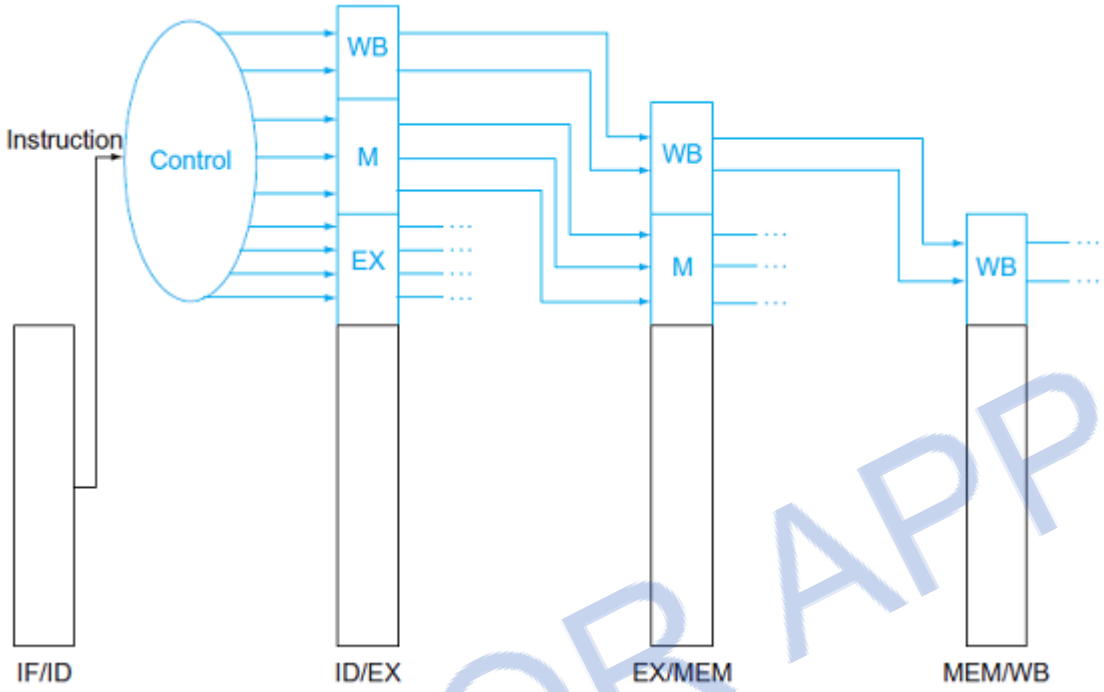


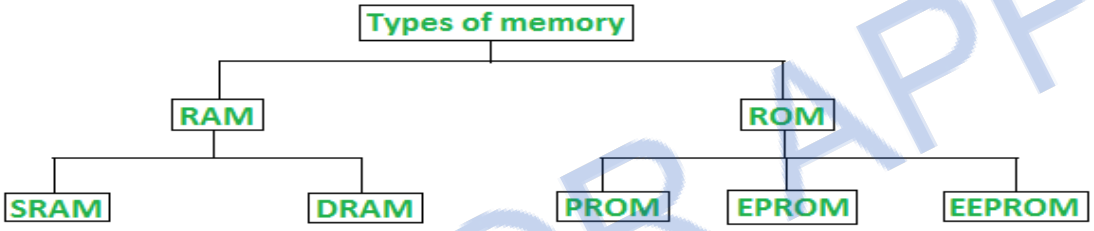
FIGURE 3.27: The control lines for the final three stages.

❖ Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information. Since the control lines start with the EX stage, we can create the control information during instruction decode. Figure 3.27 above shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline.

Memory - Introduction

Computer memory is the storage space in the computer, where data is to be processed and instructions required for processing are stored. The memory is divided into large number of small parts called cells. Each location or cell has a unique address, which varies from zero to memory size minus one.

Computer memory is of two basic type – Primary memory / Volatile memory and Secondary memory / non-volatile memory. Random Access Memory (RAM) is volatile memory and Read Only Memory (ROM) is non-volatile memory.



Classification of computer memory

MEMORY HIERARCHY

The **memory hierarchy** separates computer storage into a hierarchy based on response time. Since response time, complexity, and capacity are related, the levels may also be distinguished by their performance and controlling technologies. Memory hierarchy affects performance in computer architectural design, algorithm predictions, and lower level programming constructs involving locality of reference.

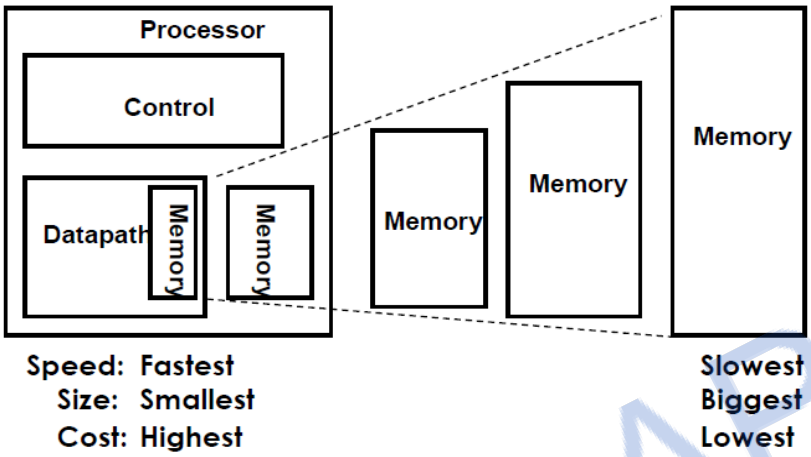
The Principle of Locality: Program likely to access a relatively small portion of the address space at any instant of time.

- Temporal Locality: Locality in Time
- Spatial Locality: Locality in Space

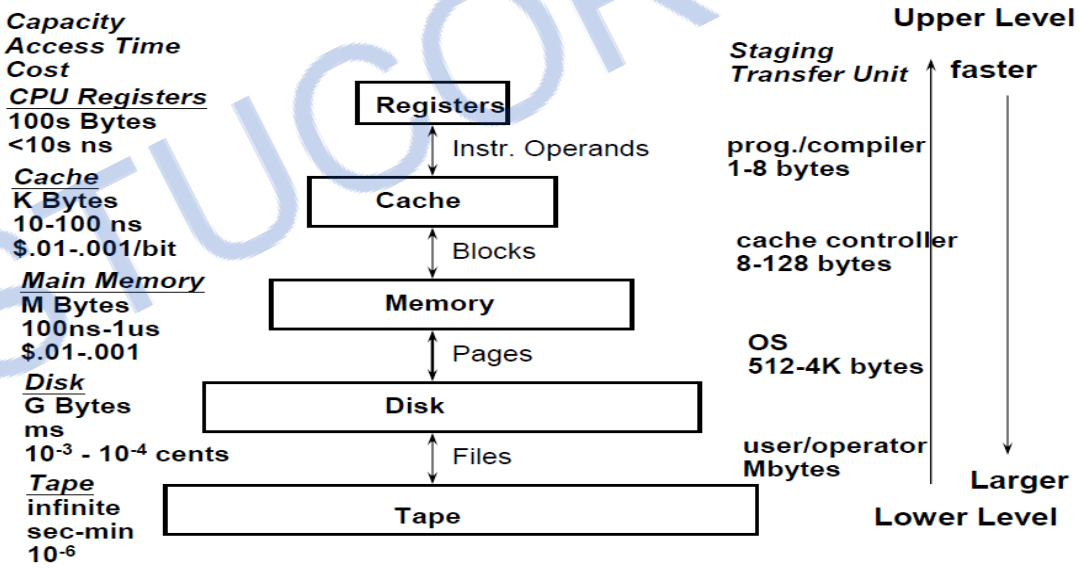
Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.

Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.

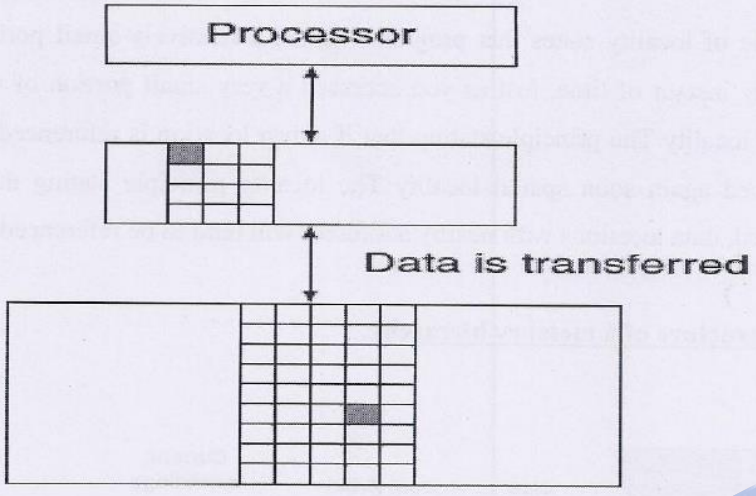
An expanded view of memory system:



Levels of Memory Hierarchy



The minimum unit of information that can be either present or not present in the two level hierarchy

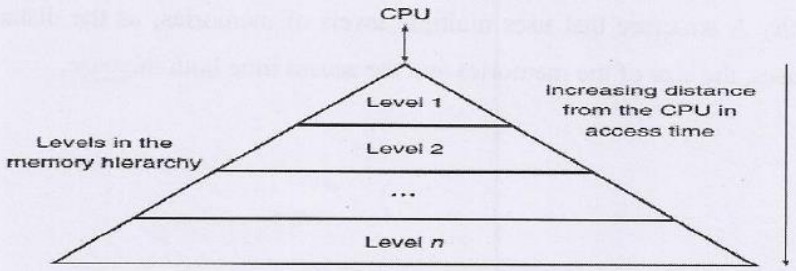


Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level block (or line) The minimum unit of information that can be either present or not present in a cache hit rate The fraction of memory accesses found in a level of the memory hierarchy.

Miss rate The fraction of memory accesses not found in a level of the memory hierarchy hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

Miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time block, transmit it from one level to the other

Structure of a memory hierarchy



to access the, insert it in the level that experienced the miss, and then pass the block to the requestor

This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. maintaining this illusion is the subject of this chapter.

Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy

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 speed difference arises from several factors 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

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 and DRAM

SRAM and DRAM are the modes of **integrated-circuit RAM** where SRAM uses transistors and latches in construction while DRAM uses capacitors and transistors. These can be differentiated in many ways, such as SRAM is comparatively faster than DRAM; hence SRAM is used for cache memory while DRAM is used for main memory.

RAM (Random Access Memory) is a kind of memory which needs constant power to retain the data in it, once the power supply is disrupted the data will be

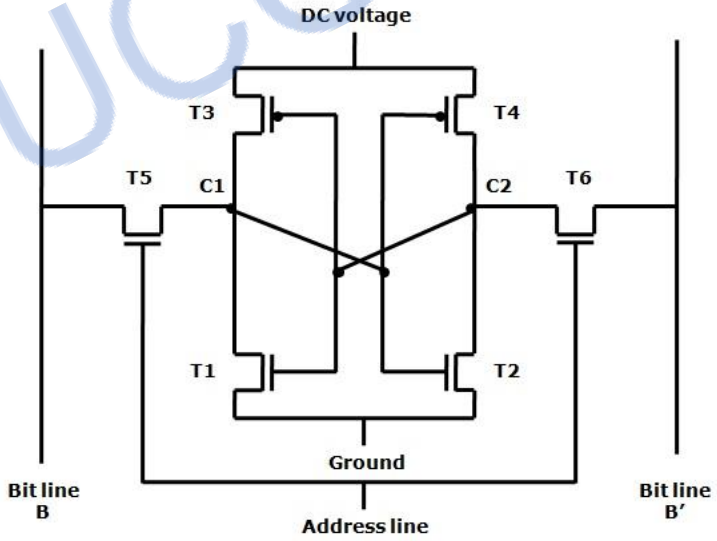
lost, that's why it is known as **volatile memory**. Reading and writing in RAM is easy and rapid and accomplished through electrical signals.

SRAM TECHNOLOGY

SRAM (Static Random Access Memory) is made up of **CMOS technology** and uses six transistors. Its construction is comprised of two cross-coupled inverters to store data (binary) similar to flip-flops and extra two transistors for access control. It is relatively faster than other RAM types such as DRAM. It consumes less power. SRAM can hold the data as long as power is supplied to it.

Working of SRAM for an individual cell:

- ✓ To generate stable logic state, four **transistors** (T1, T2, T3, T4) are organized in a cross-connected way. For generating logic state 1, node **C1** is high, and **C2** is low; in this state, **T1** and **T4** are off, and **T2** and **T3** are on.
- ✓ For logic state 0, junction **C1** is low, and **C2** is high; in the given state **T1** and **T4** are on, and **T2** and **T3** are off. Both states are stable until the direct current (dc) voltage is applied.



Static RAM (SRAM) Cell

- ✓ The SRAM **address line** is operated for opening and closing the switch and to control the T5 and T6 transistors permitting to read and write. For read operation the signal is applied to these address line then T5 and T6 gets on, and

the bit value is read from line B. For the write operation, the signal is employed to B **bit line**, and its complement is applied to B'.

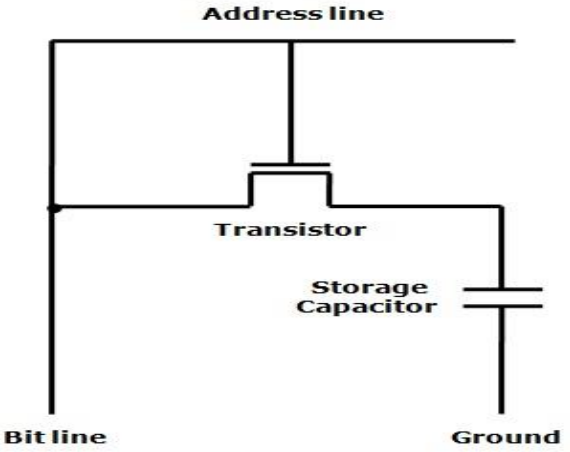
DRAM TECHNOLOGY

DRAM (Dynamic Random Access Memory) is also a type of RAM which is constructed using capacitors and few transistors. The capacitor is used for storing the data where bit value 1 signifies that the capacitor is charged and a bit value 0 means that capacitor is discharged. Capacitor tends to discharge, which result in leaking of charges.

- ✓ The dynamic term indicates that the charges are continuously leaking even in the presence of continuous supplied power that is the reason it consumes more power. To retain data for a long time, it needs to be repeatedly refreshed which requires additional refresh circuitry.
- ✓ Due to leaking charge DRAM loses data even if power is switched on. DRAM is available in the higher amount of capacity and is less expensive. It requires only a single transistor for the single block of memory.

Working of typical DRAM cell:

- ✓ At the time of reading and writing the bit value from the cell, the address line is activated. The transistor present in the circuitry behaves as a switch that is **closed** (allowing current to flow) if a voltage is applied to the address line and **open** (no current flows) if no voltage is applied to the address line. For the write operation, a voltage signal is employed to the bit line where high voltage shows 1, and low voltage indicates 0. A signal is then used to the address line which enables transferring of the charge to the capacitor.
- ✓ When the address line is chosen for executing read operation, the transistor turns on and the charge stored on the capacitor is supplied out onto a bit line and to a sense amplifier.



Dynamic RAM (DRAM) Cell

- ✓ The sense amplifier specifies whether the cell contains a logic 1 or logic 2 by comparing the capacitor voltage to a reference value. The reading of the cell results in discharging of the capacitor, which must be restored to complete the operation. Even though a DRAM is basically an analog device and used to store the single bit (i.e., 0,1).

Key Differences Between SRAM and DRAM

1. SRAM is an **on-chip** memory whose access time is small while DRAM is an **off-chip** memory which has a large access time. Therefore SRAM is faster than DRAM.
2. DRAM is available in **larger** storage capacity while SRAM is of **smaller** size.
3. SRAM is **expensive** whereas DRAM is **cheap**.
4. The **cache memory** is an application of SRAM. In contrast, DRAM is used in **main memory**.
5. DRAM is **highly dense**. As against, SRAM is **rarer**.
6. The construction of SRAM is **complex** due to the usage of a large number of transistors. On the contrary, DRAM is **simple** to design and implement.
7. In SRAM a single block of memory requires **six** transistors whereas DRAM needs just one transistor for a single block of memory.

8. Power consumption is higher in DRAM than SRAM. SRAM operates on the principle of changing the direction of current through switches whereas DRAM works on holding the charges.

Basis for comparison	SRAM	DRAM
Speed	Faster	Slower
Size	Small	Large
Cost	Expensive	Cheap
Used in	Cache memory	Main memory
Density	Less dense	Highly dense
Construction	Complex and uses transistors and latches.	Simple and uses capacitors and very few transistors.
Single block of memory requires	6 transistors	Only one transistor.
Charge leakage property	Not present	Present hence require power refresh circuitry
Power consumption	Low	High

Flash Memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

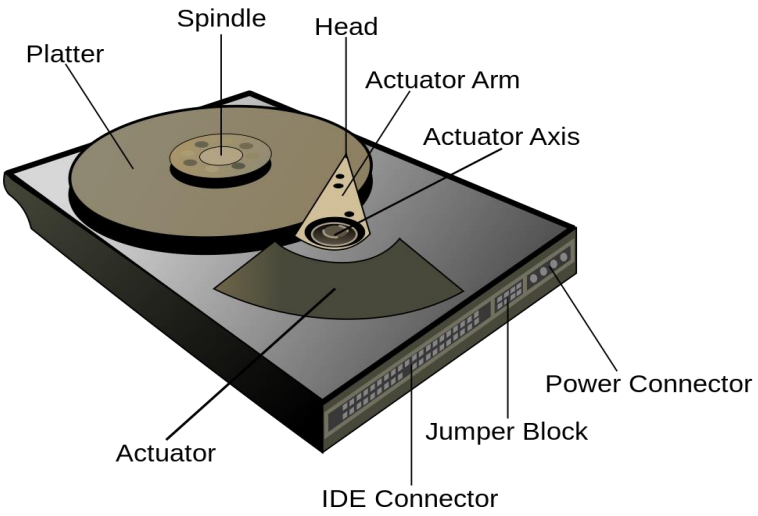
Unlike disks and DRAM, but like other EEPROM technologies, writes can wear out fl ash memory bits. To cope with such limits, most fl ash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the fl ash. Such wear leveling lowers the potential performance of fl ash, but it is needed unless higher level soft ware monitors block wear. Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

Disk Memory

Track One of thousands of concentric circles that makes up the surface of a magnetic disk.

Sector One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk. a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a ***read-write head*** is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface. Each disk surface is divided into concentric circles, called tracks. There are typically tens of thousands of tracks per surface. Each track is in turn divided into sectors that contain the information; each track may have thousands of sectors.

Sectors are typically 512 to 4096 bytes in size. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code a gap, the sector number of the next sector, and so on. The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces. Seek the process of positioning a read/write head over the proper track on a disk.

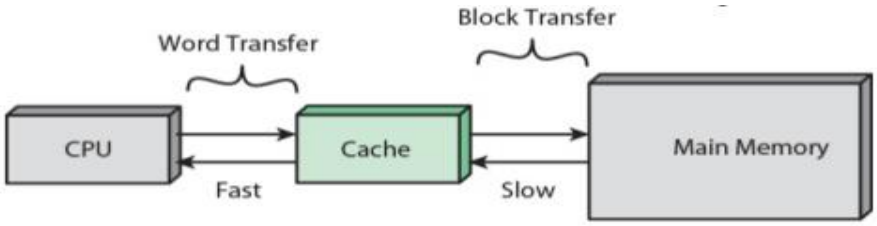


Rotational latency Also called rotational delay. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM} / \left(60 \frac{\text{seconds}}{\text{minute}}\right)}$$
$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

CACHE MEMORY:

- A Cache is a small and very fast temporary storage memory. It is designed to speed up the transfer of data and instructions. It is located inside or close to the CPU chip. It is faster than RAM and the data/instructions that are most recently or most frequently used by CPU are stored in cache.
- The data and instructions are retrieved from RAM when CPU uses them for the first time. A copy of that data or instructions is stored in cache. The next time the CPU needs that data or instructions, it first looks in cache. If the required data is found there, it is retrieved from cache memory instead of main memory. It speeds up the working of CPU.
- The cache has a significantly shorter access time than the main memory due to the applied faster but more expensive implementation technology. If information fetched to the cache memory is used again, the access time to it will be much shorter than in the case if this information were stored in the main memory and the program will execute faster.



(a) Single cache

- Time efficiency of using cache memories results from the locality of access to data that is observed during program execution. We observe here time and space locality:

The Principle of Locality

The Principle of Locality states that Program access a relatively small portion of the address space at any instant of time.

• Example: 90% of time in 10% of the code

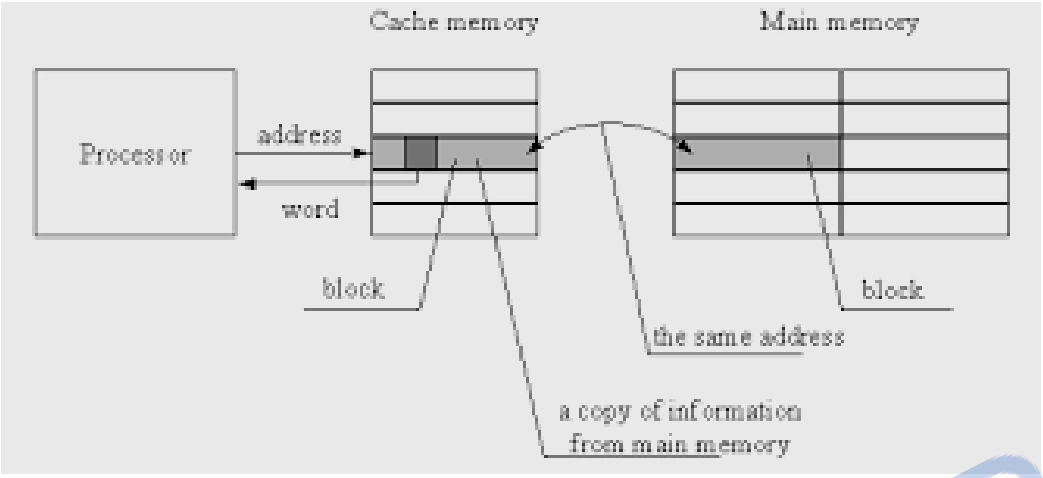
Two Different Types of Locality

Temporal Locality / Time locality (Locality in Time): It consists in a tendency to use many times the same instructions and data in programs during neighbouring time intervals, i.e., If an item is referenced, it will tend to be referenced again soon.

Spatial Locality/ Space locality (Locality in Space): It is a tendency to store instructions and data used in a program in short distances of time under neighbouring addresses in the main memory. i.e., if an item is referenced, items whose addresses are close by tend to be referenced soon.

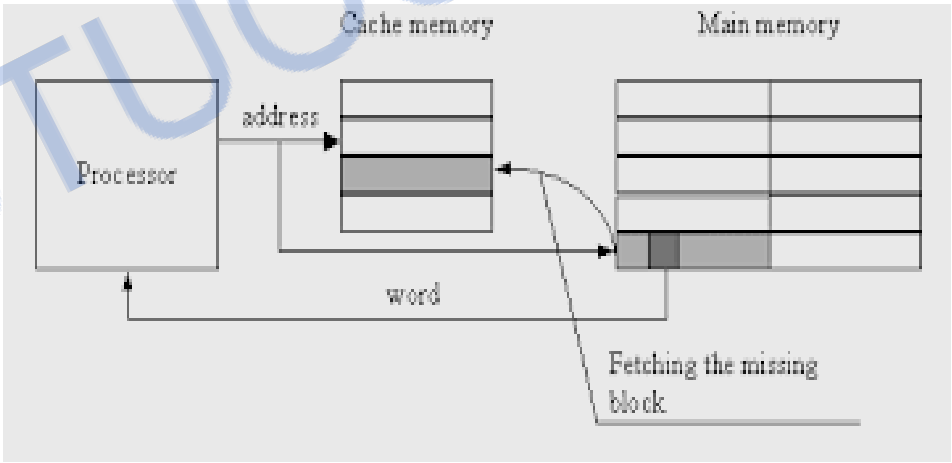
- Due to these localities, the information loaded to the cache memory is used several times and the execution time of programs is much reduced. Cache can be implemented as a multi-level memory. A cache memory is maintained by a special processor subsystem called **cache controller**.
- If there is a cache memory in a computer system, then at each access to a main memory address in order to fetch data or instructions, processor hardware sends the address first to the cache memory. The cache control unit checks if the requested information resides in the cache. If so, we have a "hit" and the

requested information is fetched from the cache. The actions concerned with a read with a hit are shown in the figure below.



Read implementation in cache memory on hit

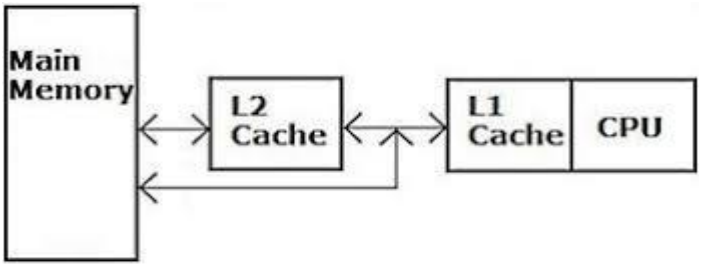
- If the requested information does not reside in the cache, we have a "miss" and the necessary information is fetched from the main memory to the cache and to the requesting processor unit. The information is not copied in the cache as single words but as a larger block of a fixed volume. The actions executed in a cache memory on "miss" are shown below.



Read implementation in cache memory on miss

If there are two cache levels, then on "miss" at the first level, the address is transferred in a hardwired way to the cache at the second level. If at this level a "hit" happens, the block that contains the requested word is fetched from the second level cache to the first level cache. If a "miss" occurs also at the second

cache level, the blocks containing the requested word are fetched to the cache memories at both levels. The size of the cache block at the first level is from 8 to several tens of bytes (a number must be a power of 2). The size of the block in the second level cache is many times larger than the size of the block at the first level.



The cache memory can be connected in different ways to the processor and the main memory:

- As an additional subsystem connected to the system bus that connects the processor with the main memory,
- As a subsystem that intermediates between the processor and the main memory,
- As a separate subsystem connected with the processor, in parallel regarding the main memory.

Categories of Cache Misses

We can subdivide cache misses into one of three categories:

- A **compulsory miss** (or **cold miss**) : It is also known as cold start misses or first references misses. These misses occur when the first access to a block happens. Block must be brought into the cache.
- **A conflict miss:** It is also known as collision misses or interference misses. These misses occur when several blocks are mapped to the same set or block frame. These misses occur in the set associative or direct mapped block placement strategies.
- **A capacity miss:** These misses occur when the program working set is much larger than the cache capacity. Since Cache cannot contain all blocks needed for program execution, so cache discards these blocks.

MEASURING AND IMPROVING CACHE PERFORMANCE

We begin by examining ways to measure and analyze cache performance. 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.

Measuring the Cache Performance

We assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus

CPU Time = (CPU execution clock cycles + Memory-stall clock cycles) x 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 write:

$$\text{Memory-stall clock cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$

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, which usually require that we fetch the block before continuing the write, 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:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Since the write buffer stalls depend on the proximity of writes, and not just the frequency, it is not possible to give a simple equation to compute such stalls.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. In most write-through cache organizations, the read and write miss penalties are the same. 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:

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

CACHE MAPPING TECHNIQUES

Memory mapping is the (complex) process that associates an address value (usually a 32 or 64 bits number) to some existing physical location in the hardware. This location can be in RAM, in a cache of some level, or even on the hard disk! During program execution, data can move from one location to another, and possibly be duplicated.

Mapping Function

The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

The different Cache mapping techniques are as follows:-

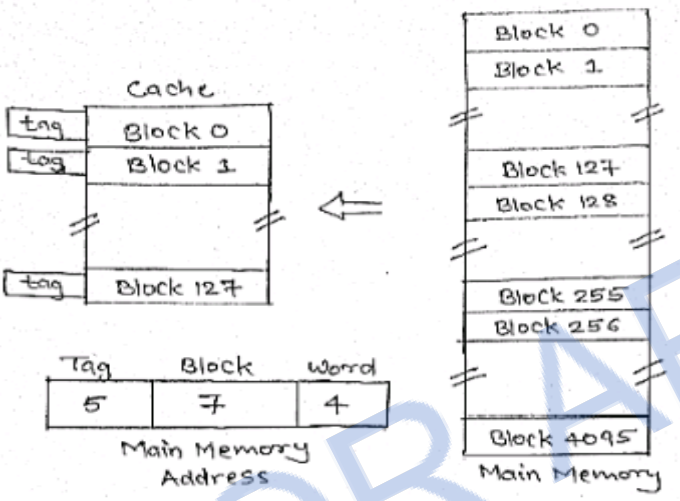
1) Direct Mapping

2) Associative Mapping

3) Set Associative Mapping

Consider a cache consisting of 128 blocks of 16 words each, for total of 2048(2K) works and assume that the main memory is addressable by 16 bit address. Main memory is 64K which will be viewed as 4K blocks of 16 works each.

(1) Direct Mapping:-

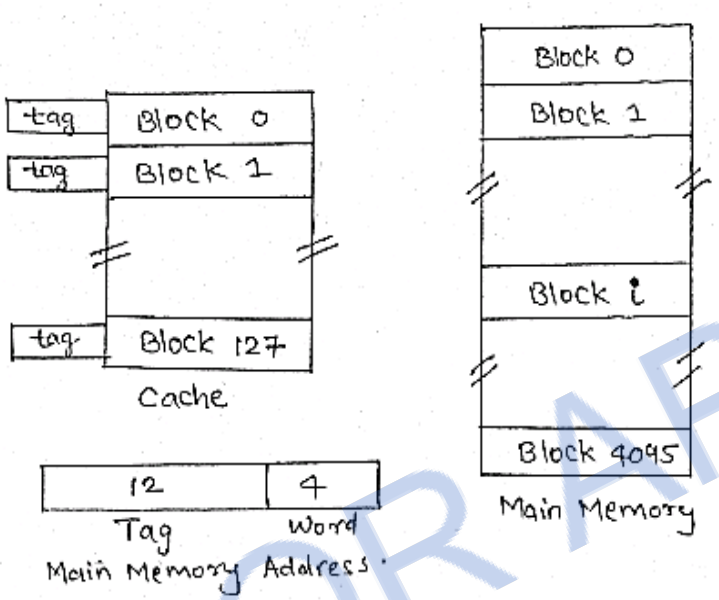


- The simplest way to determine cache locations in which store Memory blocks is direct Mapping technique.
- In this block J of the main memory maps on to block J modulo 128 of the cache. Thus main memory blocks 0,128,256,….is loaded into cache is stored at block 0. Block 1,129,257,….are stored at block 1 and so on.
- Placement of a block in the cache is determined from memory address. Memory address is divided into 3 fields, the lower 4-bits selects one of the 16 words in a block.
- When new block enters the cache, the 7-bit cache block field determines the cache positions in which this block must be stored.
- The higher order 5- in cache. They identify which of the 32 blocks that are mapped into this cache bits of the memory address of the block are stored in 5 tag bits associated with its location position are currently resident in the cache.

Advantages: It is easy to implement.

Drawbacks: 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. Contention is resolved by allowing the new block to overwrite the currently resident block. This method is not very flexible.

(2) Fully Associative Mapping:-

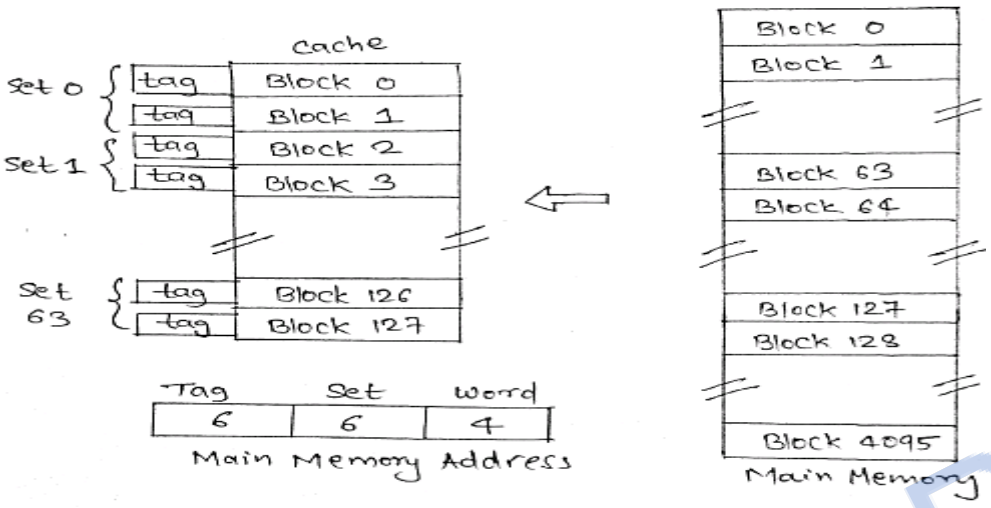


- This is more flexible mapping method, in which main memory block can be placed into any cache block position.
- In this, 12 tag bits are required to identify a memory block when it is resident in the cache.
- The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see, if the desired block is present. This is known as Associative Mapping technique.
- Cost of an associated mapped cache is higher than the cost of direct-mapped because of the need to search all 128 tag patterns to determine whether a block is in cache. This is known as associative search.

(3) Set-Associated Mapping:-

- It is the combination of direct and associative mapping technique.
- Cache blocks are grouped into sets and mapping allow block of main memory reside into any block of a specific set. Hence contention problem of direct

mapping is eased, at the same time; hardware cost is reduced by decreasing the size of associative search.



- For a cache with two blocks per set. In this case, memory block 0, 64, 128,…..,4032 map into cache set 0 and they can occupy any two block 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 bits of address must be associatively compared to the tags of the two blocks of the set to check if desired block is present. This is known as two way associative search.

Advantages:

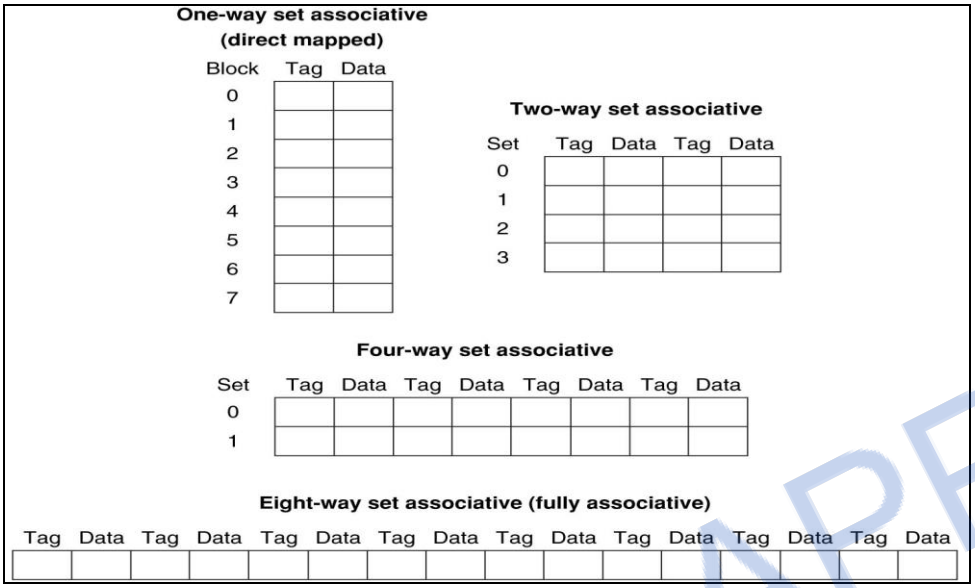
The contention problem of the direct-mapping is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search.

M - Way Set Associativity:

We can also think of all block placement strategies as a variation on set associativity. The following figure shows the possible associativity structures for an eight-block cache.

- A direct-mapped cache is simply a one-way set-associative cache: each cache entry holds one block and each set has one element.

- A fully-associative cache with m entries is simply an m-way set-associative cache: it has one set with m blocks, and an entry can reside in any block within that set.



The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The main disadvantage is the potential increase in the hit time.

VIRTUAL MEMORY

Introduction

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- Users would be able to write programs for an extremely large virtual-address space, simplifying the programming task.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.
- Virtual memory is the separation of user logical memory from physical

memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

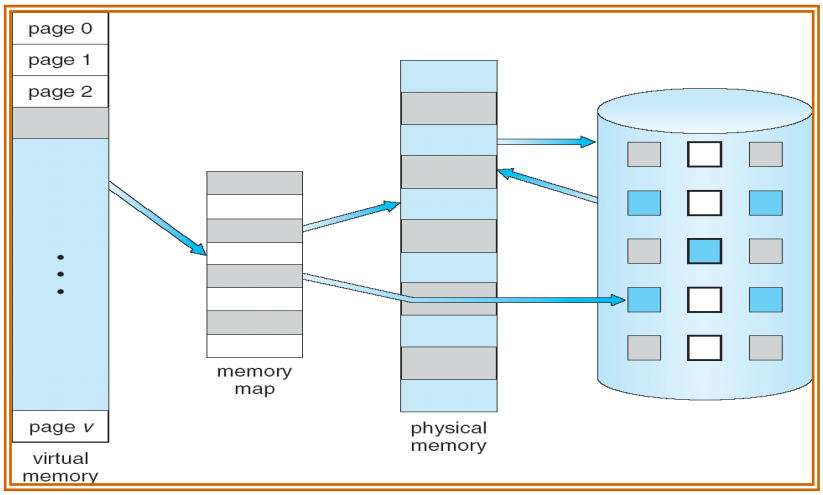


Diagram showing virtual memory that is larger than physical memory

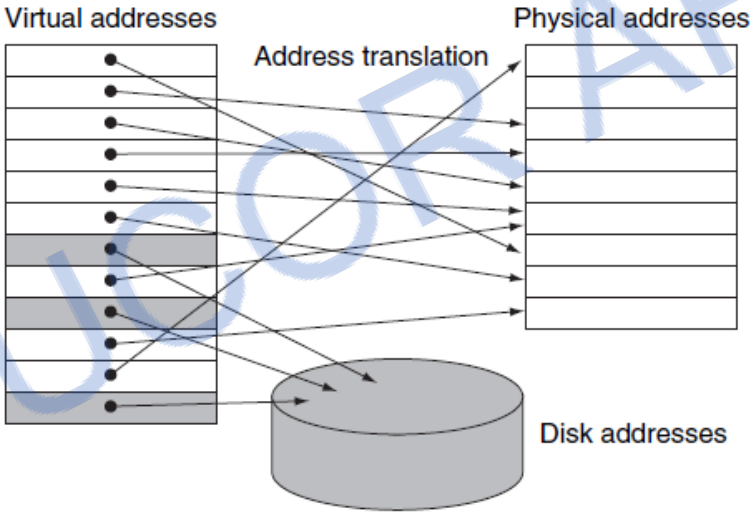


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).

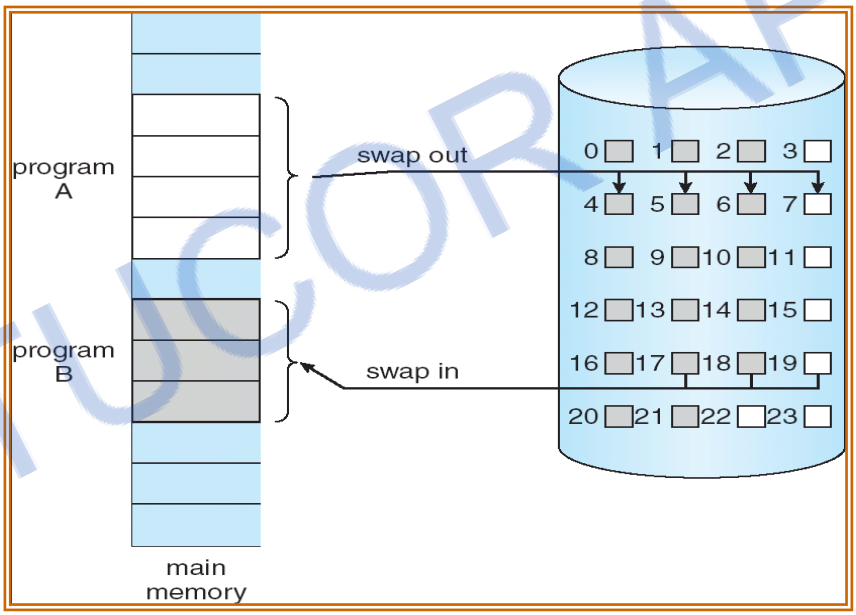
Demand Paging

- Virtual memory is commonly implemented by demand paging.
- **A demand-paging system is similar to a paging system with swapping. Pages are loaded only on demand and not in advance.**
- Processes reside on secondary memory (which is usually a disk). When the process is to be executed, it is swapped in to memory. Rather than swapping the entire process into memory, however, use a **lazy swapper**. A lazy swapper

never swaps a page into memory unless that page will be needed.

Basic Concepts

Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.



Diag: Transfer of a paged memory to contiguous disk space.

- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page-fault trap.
- The process can now access the page as though it had always been in memory. When the disk read is complete, we modify the internal table kept with the

process and the page table to indicate that the page is now in memory.

Pure demand paging:

Never bring a page into memory until it is required. The hardware to support demand paging is the same as the hardware for paging and swapping:

Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**.

PAGING IN VIRTUAL MEMORY

- Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.

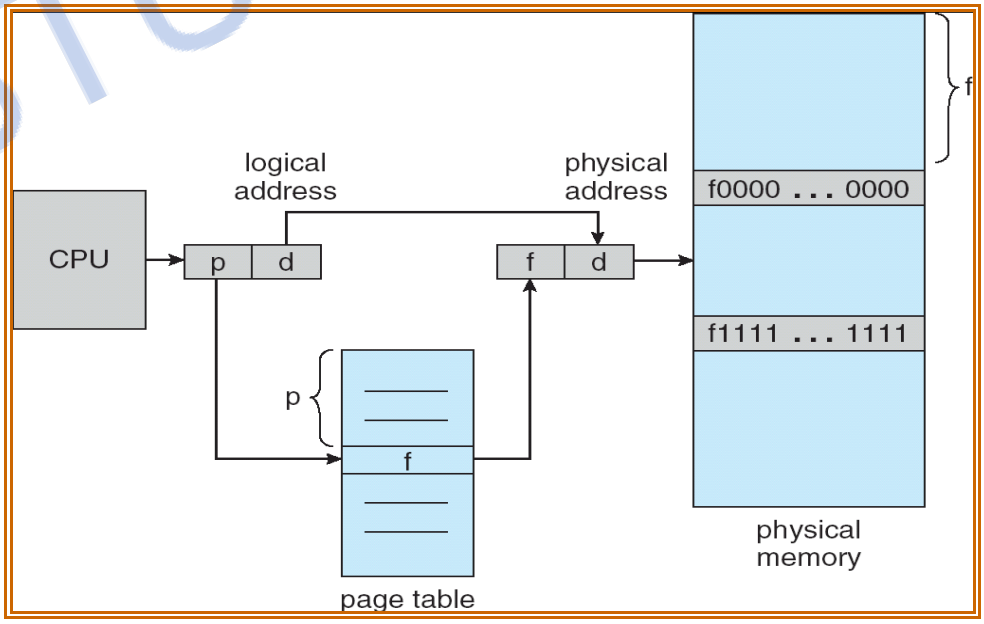
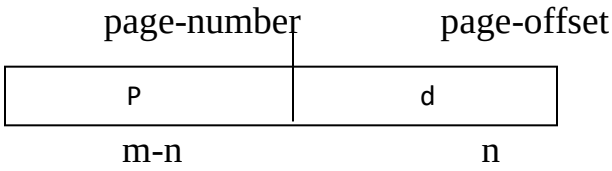


Fig: Paging Hardware

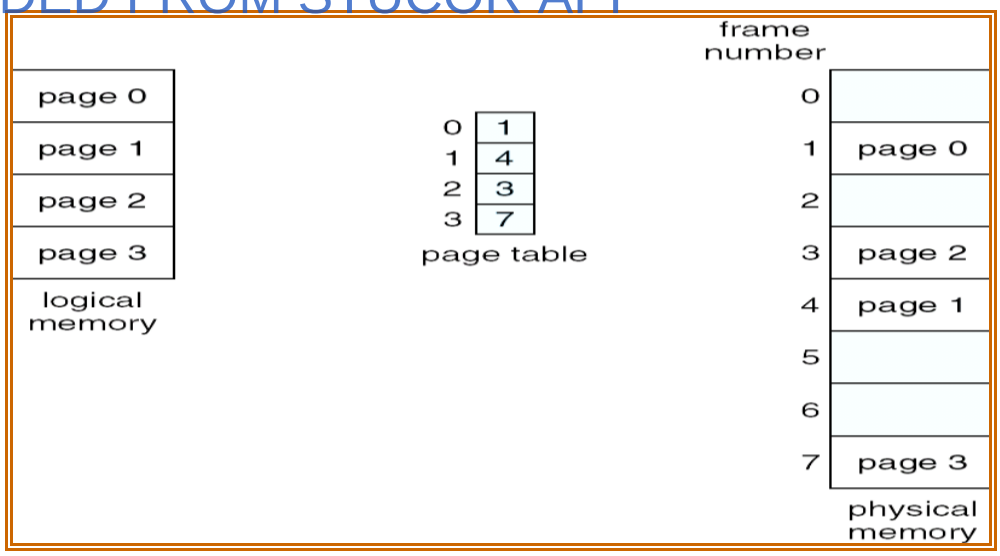
Basic Method:

Physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.



Where **p** is an index into the page table and **d** is the displacement within the page.

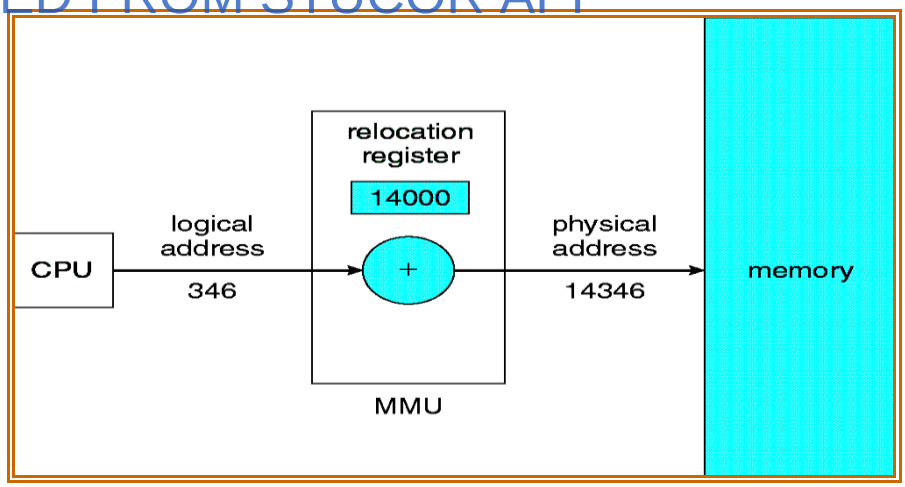
- The hardware support for paging is illustrated in the above Figure. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in the Figure given below.
- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.



Diag: Paging model of logical and physical memory

LOGICAL Versus PHYSICAL ADDRESS SPACE

- Logical address is generated by the CPU. Physical address is the address of main memory and it is loaded in to the memory address register.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses. Usually the logical address is referred as a virtual address.
- The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-address space. Thus, in the execution-time address-binding scheme, the logical and physical-address spaces differ.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).
- As illustrated in the given figure, the base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.



Diag: Address translation using a relocation register in MMU

- The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it to other addresses-all as the number 346. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.

ADDRESS TRANSLATION – EXAMPLE

Consider the memory shown in the following Figure. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0). Logical address 13 maps to physical address 9.

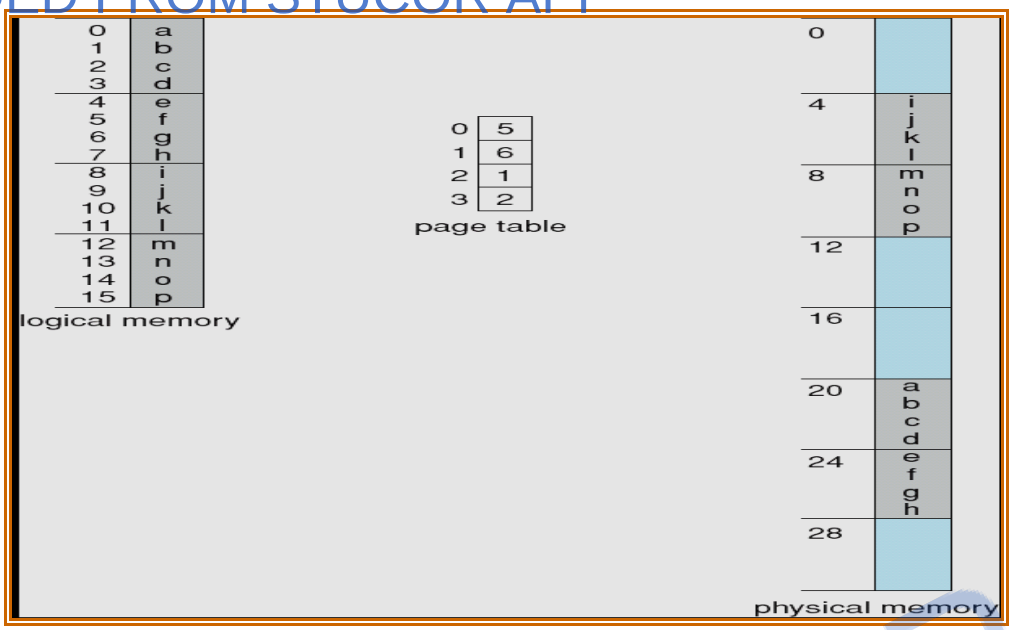


Fig: Paging example for 32-byte memory with 4-byte pages

- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today pages typically are between 4 KB and 8 KB, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses 8 KB and 4 MB page sizes, depending on the data stored by the pages. Researchers are now developing variable on-the-fly page-size support. Each page-table entry is usually 4 bytes long, but that size can vary as well.

TLB (TRANSLATION LOOK-ASIDE BUFFER)

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

The page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers.

The standard solution to this problem is to use a special, small, fast lookup hardware cache, called translation look-aside buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: L a key (or tag) and a value. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

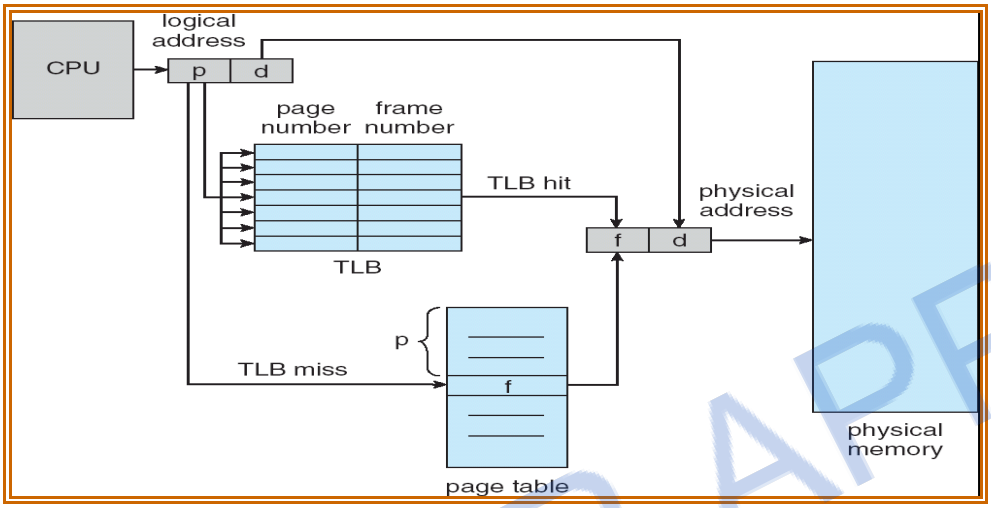


Fig: Paging Hardware with TLB

The TLB is used with page tables in the following way.

- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.

The percentage of times that a particular page number is found in the TLB is called the hit ratio. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped- memory access takes 120 nanoseconds when the page number is in the TLB.

To find the effective memory-access time, we must weigh each case by its probability:

Effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.

PROTECTION in TLB

Protection bits are kept in the page table. The valid-invalid bit scheme can be used for this purpose. When this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk.

- The page-table entry for a page that is brought into memory is set as usual, but the page table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in the following figure.

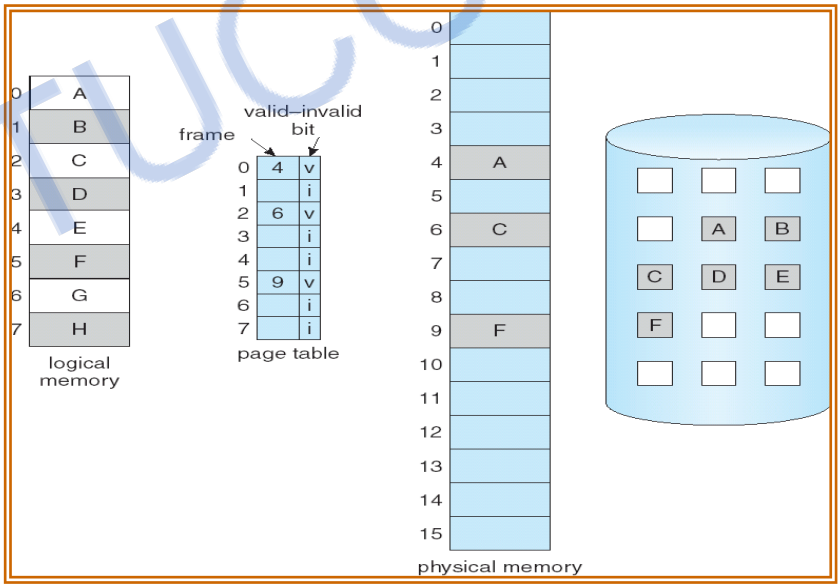
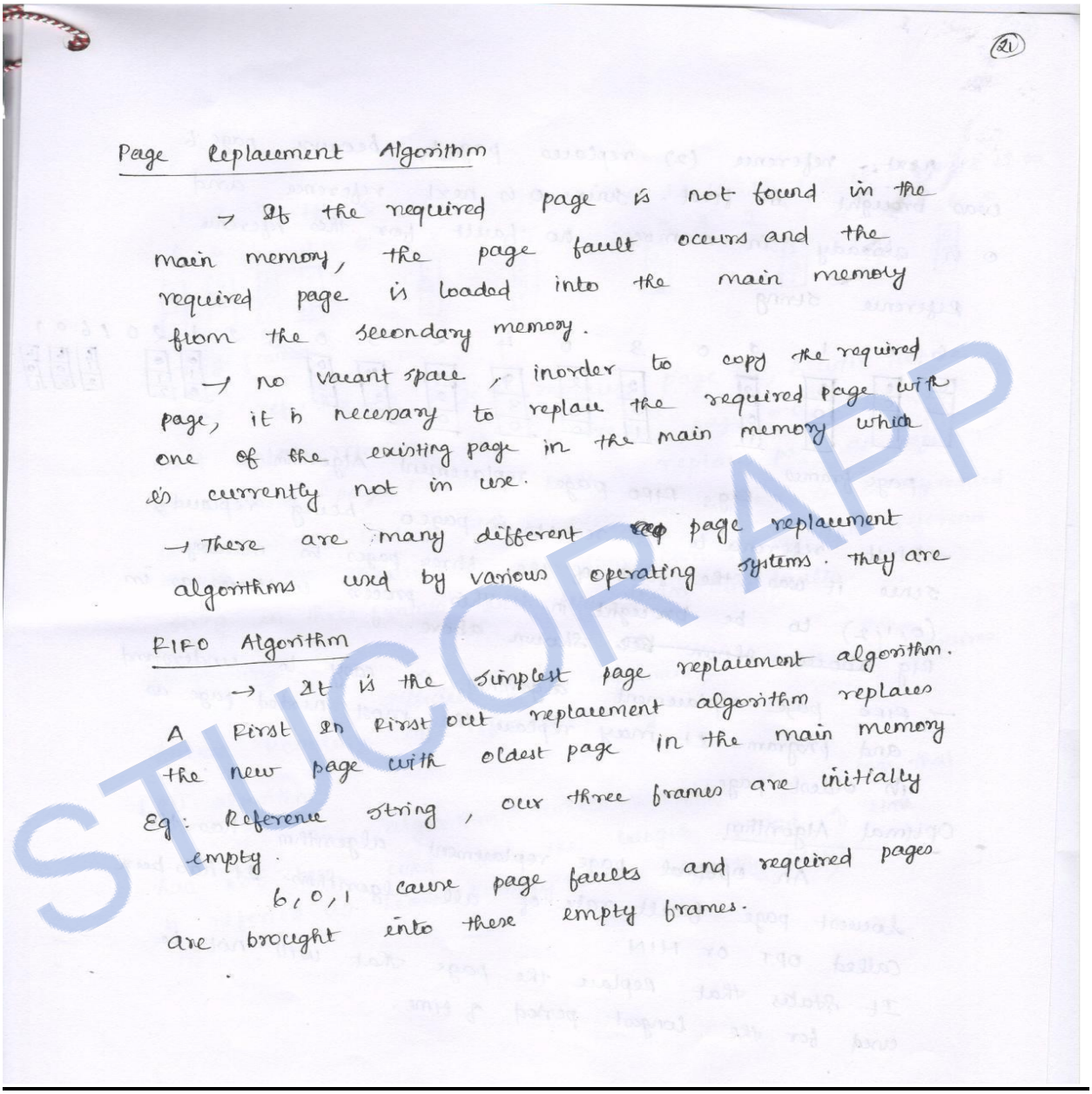


Fig: Valid (v) or invalid (i) bit in a page table.

- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into

memory, rather than an invalid address error as a result of an attempt to use an illegal memory address.

PAGE REPLACEMENT ALGORITHMS



(21)

Page Replacement Algorithm

→ If the required page is not found in the main memory, the page fault occurs and the required page is loaded into the main memory from the secondary memory.

→ no vacant space, inorder to copy the required page, it is necessary to replace the required page with one of the existing page in the main memory which is currently not in use.

→ There are many different page replacement algorithms used by various operating systems. they are

FIFO Algorithm

→ It is the simplest page replacement algorithm. A First In First out replacement algorithm replaces the new page with oldest page in the main memory

Eg: Reference string, our three frames are initially empty.

6, 0, 1 cause page faults and required pages are brought into these empty frames.

Next reference (2) replaces page 6, because page 6 was brought in first. Since 0 is next reference and 0 is already in memory, no fault for this reference

Reference String

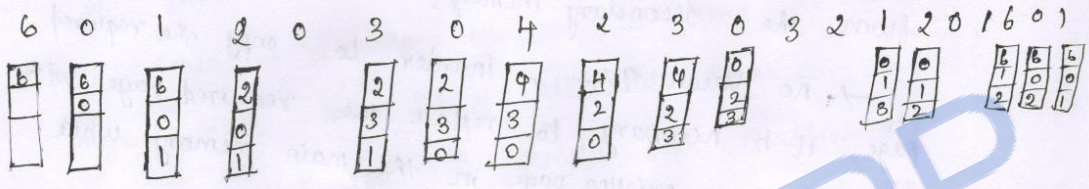


Fig: FIFO page replacement Algorithm

→ First reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, 2) to be brought in. This process continues as in Fig above ~~shown below~~ shown above.

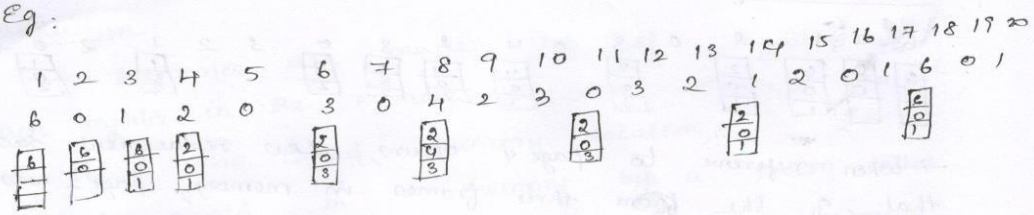
→ FIFO page replacement algorithm is easy to understand and program. It may replace the most needed page as it is oldest page.

Optimal Algorithm

An optimal page replacement algorithm has the lowest page fault rate of all algorithms. It has been called OPT or MIN

It states that Replace the page that will not be used for the longest period of time.

Eg:



page frames

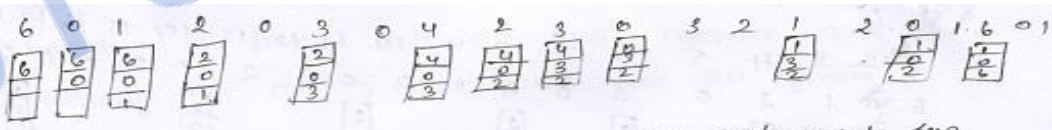
The reference to page 2 replaces page 6, 6 will not be used untill reference 18, whereas page 0 will be used at 5 and page 1 at 14. page 3 replaces page 1, as page 1 will be the last of three pages in memory to be referenced again. No replacement algorithm can process the reference string in three frames with less than nine faults.

Disadvantage

It is difficult to implement, because it requires future knowledge of reference string.

LRU algorithm

The algorithm which replaces the page that has not been used for the longest period of time is referred as least Recently Used algorithm (LRU).



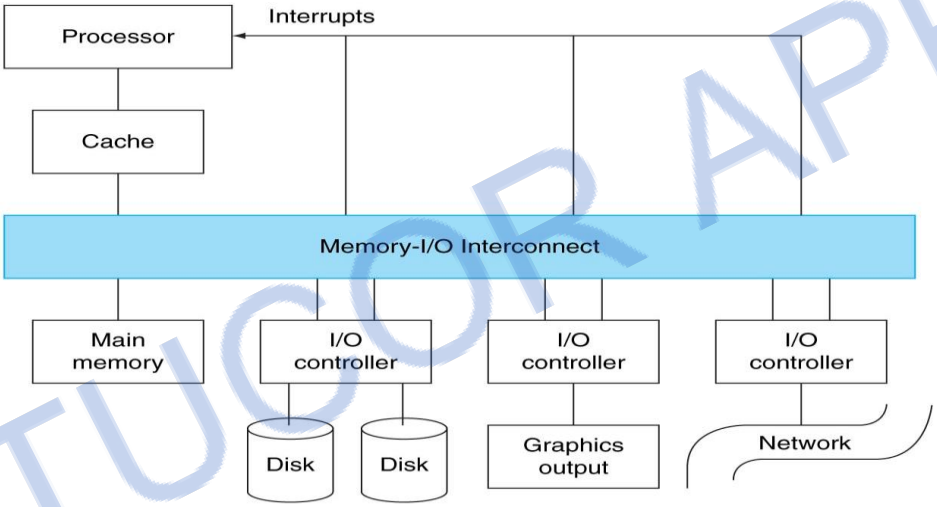
→ When reference to page 4 occurs, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Recently used page is 0 and just before that page 3 was used. LRU algorithm replaces page 2 Not knowing that page 2

→ When page 2 fault, LRU replaces page 3, among (0,3,9) page 3 is least recently used

→ LRU with 12 faults is still much better than FIFO replacement with 15.

INPUT / OUTPUT SYSTEM

- The main data-processing functions of a computer involve its CPU and external memory. The CPU fetches instructions and data from memory, processes them, and eventually stores the results back in memory.
- The other system components like secondary memory, user interface devices, and so on constitute the input/output (I/O) system. One of the basic features of a computer is its ability to exchange data with other devices. The data transfer rate of peripherals is much slower than that of the memory or CPU. The I/O subsystem provides the mechanism for communications between CPU and the outside world.



- The connection between the I/O devices, processor, and memory are historically called buses, although the term mean shared parallel wires and most I/O connections today are closer to dedicated serial lines. Communication among the devices and the processor uses both interrupts and protocols on the interconnection.

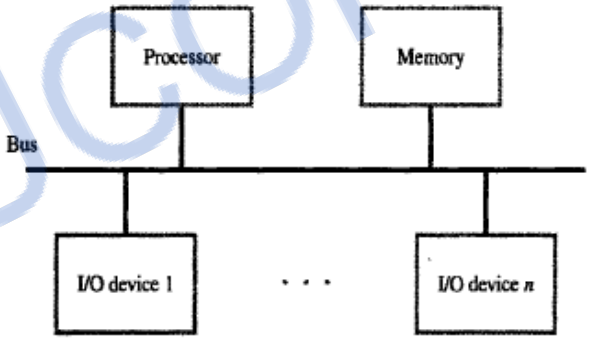
I/O devices are incredibly diverse. Three characteristics are useful in organizing this wide variety:

1. **Behavior:** Input (read once), output (write only, cannot be read), or storage (can be read and usually rewritten).

2. **Partner:** Either a human or a machine is at the other end of the I/O device, either feeding data on input or reading data on the output.
3. **Data rate:** The peak rate at which data can be transferred between the I/O device and the main memory or processor. It is useful to know the maximum demand the device may generate when designing an I/O system.

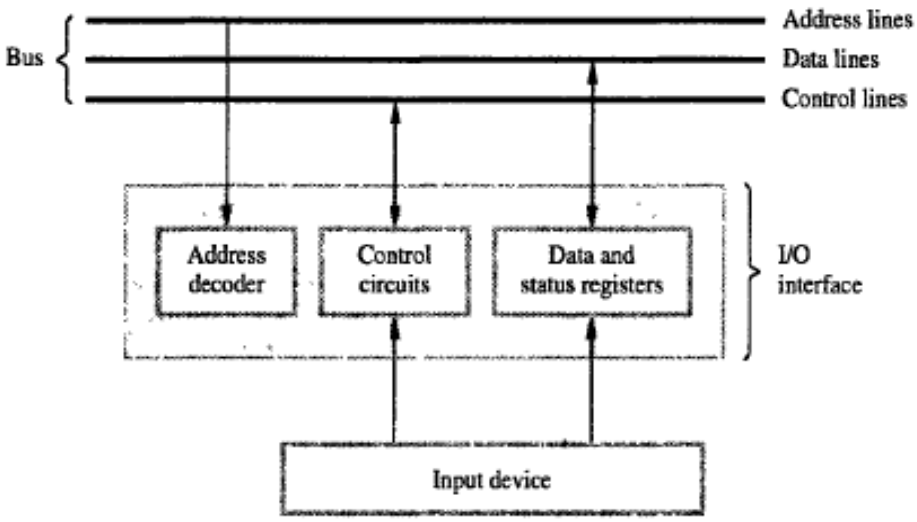
Accessing I/O Devices

A simple arrangement to connect I/O devices to a computer is to use a single bus arrangement. The bus enables all the devices connected to it to exchange information. Typically it consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the requested data are transferred over the data lines.



I/O Interface

- The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.
- The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor.



- The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses.
- I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. Also, we must make sure that an input character is read only once.
- For an input device such as keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. A similar procedure can be used to control output operations using an output status flag, SOUT.

I/O Control Methods

Input-Output operations are distinguished by the extent to which the CPU is involved in their execution. I/O operation refers to a data transfer between an IO device and memory, or between an IO device and the CPU.

Commonly used mechanisms for implementing IO operations

There are three commonly used methods used for implementing IO operations. They are

1. **Programmed I/O**

If I/O operations are completely controlled by the CPU, i.e., the CPU executes programs that initiate, direct, and terminate the IO operations, then the computer is said to be using programmed IO. This type of IO control can be implemented with little or no special hardware, but causes the CPU to spend a lot of time performing relatively trivial IO-related functions.

2. **Direct Memory Access (DMA)**

A modern hardware design enables an IO device to transfer a block of information to or from memory without CPU intervention. This task requires the IO device to generate memory addresses and transfer data to or from the bus connecting it to memory via its interface controller; in other words, the IO device must be able to act as a bus master.

The CPU is still responsible for initiating each block transfer. The IO device interface controller can then carry out the transfer without further program execution by the CPU. The CPU and IO controller interact only when the CPU must yield control of the memory bus to the IO controller in response to requests from the latter. This level of IO control is called Direct Memory Access (DMA) and the IO device interface control circuit is called a DMA Controller.

3. **Interrupts**

The DMA controller can also be provided with circuits enabling it to request service from the CPU, that is, execution of a specific program to service an IO device. This type of request is called an Interrupt and it frees the CPU from the task of periodically testing the status of IO devices.

Unlike a DMA request, which merely requests temporary access to the system bus, an interrupt request causes the CPU to switch programs by saving its previous program state and transferring control to a new interrupt-handling program. After

the interrupts has been serviced, the CPU can resume execution of the interrupted program.

DIRECT MEMORY ACCESS (DMA)

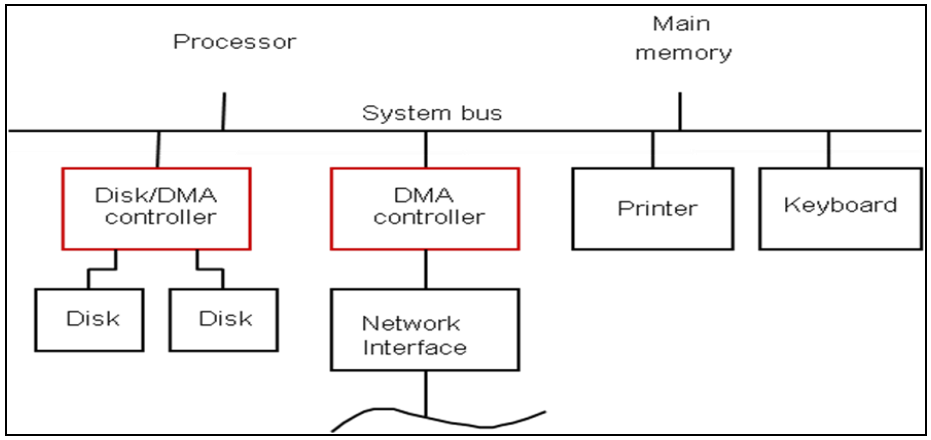
A special control unit is provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called direct memory access, or DMA.

(OR)

DMA stands for "Direct Memory Access" and is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. While most data that is input or output from your computer is processed by the CPU, some data does not require processing, or can be processed by another device.

In these situations, DMA can save processing time and is a more efficient way to move data from the computer's memory to other devices. In order for devices to use direct memory access, they must be assigned to a DMA channel. Each type of port on a computer has a set of DMA channels that can be assigned to each connected device.

- Transfer of data between a fast storage device and memory is limited by the speed of CPU
- Remove CPU from the path of communication and the technique is DMA



- DMA transfers are performed by a control circuit that is part of the I/O device interface. We refer to this circuit as a DMA controller. 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 all the bus signals that control data transfer. Since it has to transfer blocks of data, the DMA controller must increment the memory address for successive words and keep track of the number of transfers.
- Although a DMA controller can transfer data without intervention by the processor, its operation must be under the control of a program executed by the processor.
- To initiate the transfer of a block of words, the processor sends the starting address, the number of words in the block, and the direction of the transfer. On receiving this information, the DMA controller proceeds to perform the requested operation. When the entire block has been transferred, the controller informs the processor by raising an interrupt signal.
- While a DMA transfer is taking place, the program that requested the transfer cannot continue, and the processor can be used to execute another program. After the DMA transfer is completed, the processor can return to the program that requested the transfer. I/O operations are always performed by the operating system of the computer in response to a request from an application program.
- The OS is also responsible for suspending the execution of one program and starting another. Thus, for an I/O operation involving DMA, the OS puts the program that requested the transfer in the Blocked state initiates the DMA operation, and starts the execution of another program. When the transfer is completed, the DMA controller informs the processor by sending an interrupt

request. In response, the OS puts the suspended program in the runnable state so that it can be selected by the scheduler to continue execution.

- **Cycle Stealing** –The DMA controller must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. This technique is referred to as cycle stealing. It allows DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU

DMA Controller

- A simple DMA controller is a standard component in modern PCs, and many bus-mastering I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus(including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as Direct Virtual Memory Access, DVMA, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

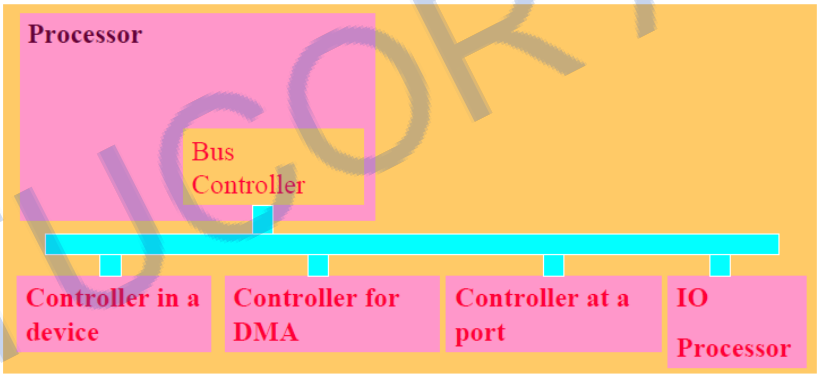
The controller is integrated into the processor board and manages all DMA data transfers. Transferring data between system memory and an I/O device requires two steps.

i. Data goes from the sending device to the DMA controller and then to the receiving device. The microprocessor gives the DMA controller the location, destination, and amount of data that is to be transferred. Then the DMA controller transfers the data, allowing the microprocessor to continue with other processing tasks. When a device needs to use the Micro Channel bus to

send or receive data, it competes with all the other devices that are trying to gain control of the bus. This process is known as **arbitration.**

ii. The DMA controller does not arbitrate for control of the BUS instead; the I/O device that is sending or receiving data (the DMA slave) participates in arbitration.

- DMA controller takes over the buses to manage the transfer directly between the I/O device and memory
- Bus Request (BR) –used by the DMA controller to request the CPU to claim or give up control of the buses.
- CPU activates bus grant to inform the external DMA that the buses are in high impedance state.
- **Burst transfer** –block sequence consisting of memory words is transferred in a continuous bus when DMA controller is the master.



INTERRUPTS

- An **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.
- The processor responds by suspending its current activities, saving its state, and executing a function called an *interrupt handler* (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.

There are two types of interrupts: hardware interrupts and software interrupts.

- **Hardware interrupts** are used by devices to communicate that they require attention from the operating system. Internally, hardware interrupts are implemented using electronic alerting signals that are sent to the processor from an external device, which is either a part of the computer itself, such as a disk controller, or an external peripheral.
- For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ).
- A **software interrupt** is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed. The former is often called a trap or exception (For example a divide-by-zero exception) and is used for errors or events occurring during program execution.
- Each interrupt has its own interrupt handler. The number of hardware interrupts is limited by the number of interrupt request (IRQ) lines to the processor, but there may be hundreds of different software interrupts. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Interrupts Handling

- The ***interrupt mechanism allows devices to signal the CPU and to force execution*** of a particular piece of code.
- When an interrupt occurs, the program counter's value is changed to point to an ***interrupt handler routine (also commonly known*** as a ***device driver) that takes care of the device.***

- The interface between the CPU and I/O device includes the following signals for interrupting:
- The I/O device asserts the ***interrupt request signal when it wants service*** from the CPU; and
- The CPU asserts the ***interrupt acknowledge signal when it is ready to handle*** the I/O device's request.

The interrupt mechanism:

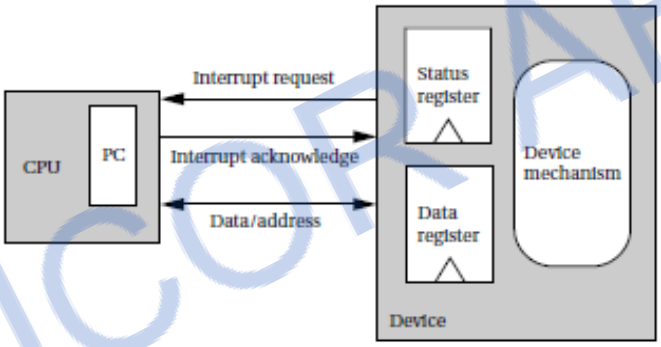


FIGURE 3.2
The interrupt mechanism.

- The *interrupt mechanism allows devices to signal the CPU and to force execution* of a particular piece of code.
- When an interrupt occurs, the program counter's value is changed to point to an *interrupt handler routine (also commonly known* as a *device driver) that takes care of the device.*
- The interface between the CPU and I/O device includes the following signals for interrupting:

  ■ the I/O device asserts the *interrupt request signal when it wants service* from the CPU; and

■ The CPU asserts the *interrupt acknowledge signal when it is ready to handle* the I/O device's request.

Priorities and Vectors

- ***Interrupt priorities*** allow the CPU to recognize some interrupts as more important than others, and
- ***Interrupt vectors*** allow the interrupting device to specify its handler.

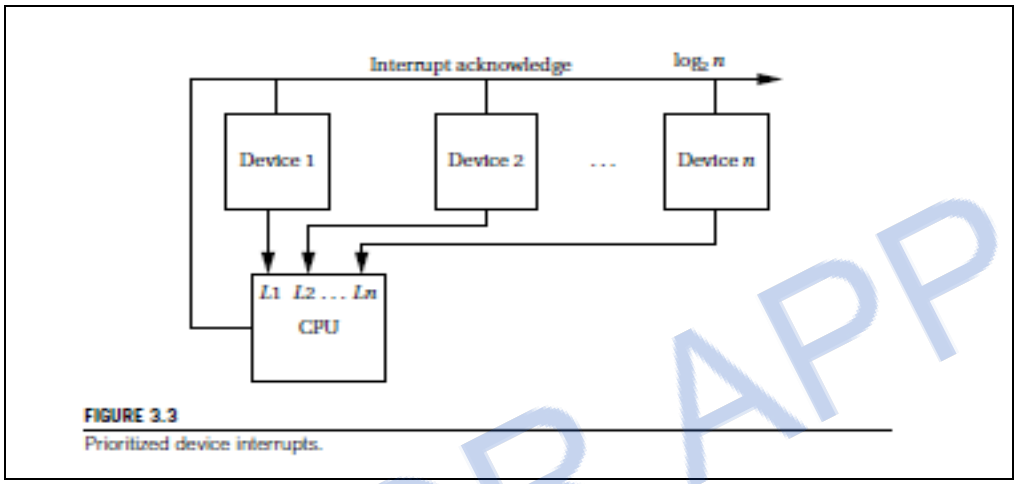


FIGURE 3.3
Prioritized device interrupts.

- The priority mechanism must ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is being handled. The decision process is known as ***masking.***
- Asking an I/O device whether it is finished by reading its status register is often called ***polling.***
- The highest-priority interrupt is normally called the ***non maskable interrupt (NMI).***
- Most CPUs provide a relatively small number of interrupt priority levels, such as eight. When several devices naturally assume the same priority. You can **combine polling with prioritized interrupts** to efficiently **handle** the devices.
- Interrupt request signals

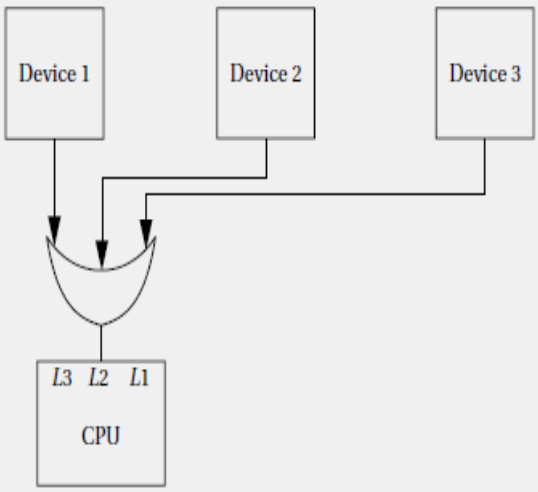


FIGURE 3.4
Using polling to share an interrupt over several devices.

- The CPU will call the interrupt handler associated with this priority; that handler does not know which of the devices actually requested the interrupt. The handler uses software polling to check the status of each device: In this example, it would read the status registers of 1, 2, and 3 to see which of them ready and requesting service is. The given example illustrates how priorities affect the order in which I/O requests are handled.

➢ **Vectors** provide flexibility in a different dimension, namely, the ability to define the interrupt handler that should service a request from a device. Figure shows the hardware structure required to support interrupt vectors. In addition to the interrupt request and acknowledge lines, additional interrupt vector lines run from the devices to the CPU.

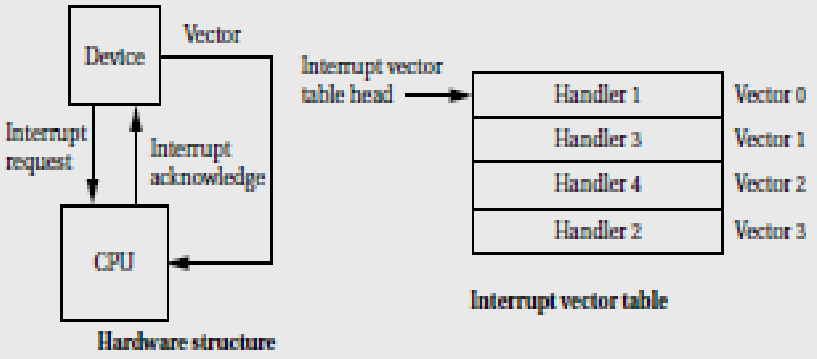


FIGURE 3.5

Interrupt vectors.

- After a device's request is acknowledged, it sends its interrupt vector over those lines to the CPU. The CPU then uses the vector number as an index in a table stored in memory as shown in Figure 3.5. The location referenced in the interrupt vector table by the vector number gives the address of the handler.
- There are two important things to notice about the interrupt vector mechanism. First, the device, not the CPU, stores its vector number. In this way, a device can be given a new handler simply by changing the vector number it sends, without modifying the system software. For example, vector numbers can be changed by programmable switches.
- The second thing to notice is that there is no fixed relationship between vector numbers and interrupt handlers. The interrupt vector table allows arbitrary relationships between devices and handlers. The vector mechanism provides great flexibility in the coupling of hardware devices and the software routines that service them.
- Most modern CPUs implement both prioritized and vectored interrupts. Priorities determine which device is serviced first, and vectors determine what routine is used to service the interrupt. The combination of the two provides a rich interface between hardware and software.

BUS STRUCTURE

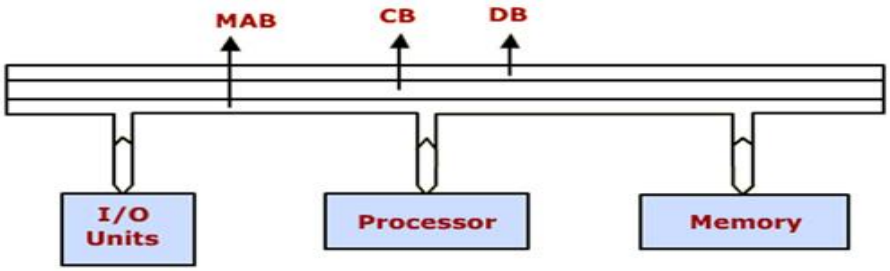
A computer system is made up of 3 major components. Central Processing Unit (CPU) that processes data, Memory Unit that holds data for processing and the Input and Output Unit that is used by the user to communicate with the computer. But how do these different components of a CPU communicate with each other? They use a special electronic communication system called the BUS. The computer bus carries lots of information using numerous pathway called circuit lines. The **System bus** consists of data bus, address bus and control bus

- ✓ **Data bus**- A bus which carries data to and from memory/IO is called as data bus
- ✓ **Address bus**- This is used to carry the address of data in the memory and its width is equal to the number of bits in the MAR of the memory. For example, if a computer memory of 64K has 32 bit words then the computer will have a data bus of 32 bits wide and the address bus of 16 bits wide.
- ✓ **Control Bus**- carries the control signals between the various units of the computer. **Ex: Memory Read/write, I/O Read/write**

Two types of Bus organizations:

- Single Bus organization
- Two bus Organization

Single Bus Architecture



Single - Bus Organization

- Three units share the single bus. At any given point of time, information can be transferred between any two units
- Here I/O units use the same memory address space (Memory mapped I/O)

- So no special instructions are required to address the I/O, it can be accessed like a memory location
- Since all the devices do not operate at the same speed, it is necessary to smooth out the differences in timings among all the devices A common approach used is to include buffer registers with the devices to hold the information during transfers

Ex: Communication between the processor and printer

Two Bus Architecture

Two Bus Organization

- Various units are connected through two independent buses
- I/O units are connected to the processor though an I/O bus and Memory is connected to the processor through the memory bus
- I/O bus consists of address, data and control bus Memory bus also consists of address, data and control bus. In this type of arrangements processor completely supervises the transfer of information to and from I/O units. All the information is first taken to processor and from there to the memory. Such kind of transfers is called as program controlled transfer.

Bus arbitration process

Multiple devices may need to use the bus at the same time so must have a way to arbitrate multiple requests.

Bus Arbitration refers to the process by which the current bus master accesses and then leaves the control of the bus and passes it to another bus requesting processor unit. The controller that has access to a bus at an instance is known as a **Bus master**.

- Bus Arbitration Mechanism between the System buses are shared between the controllers and an IO processor and multiple controllers that have to access the bus, but only one of them can be granted the bus master status at any one instance
- Bus master has the access to the bus at an instance controller and an IO processor and multiple controllers that have to access the bus, but only one of them can be granted the bus master status at any one instance
- Bus master has the access to the bus at an instance.

There are two approaches to bus arbitration:

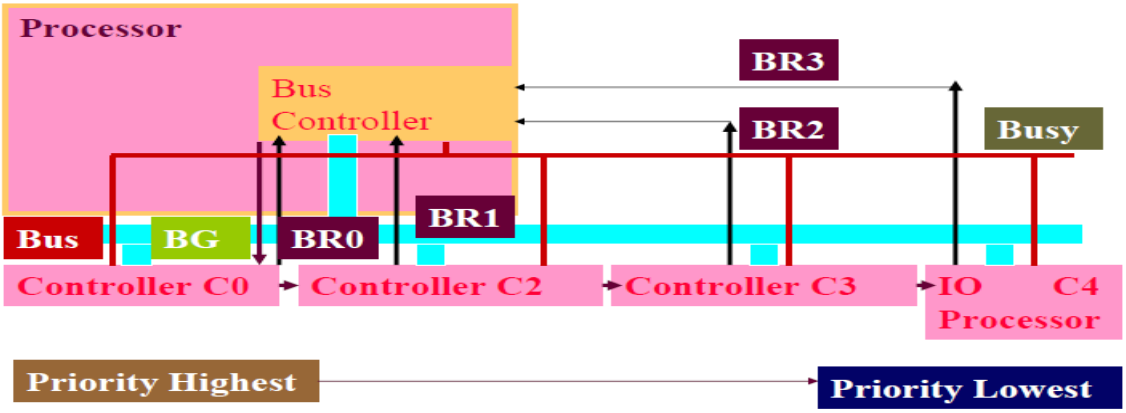
1. **Centralized bus arbitration –** A single bus arbiter performs the required arbitration.
2. **Distributed bus arbitration –** All devices participate in the selection of the next bus master.

Three methods in Centralized bus arbitration process

- **Daisy Chain method**
- **Fixed Priority or Independent Bus Requests and Grant method**
- **Polling or Rotating Priority method**

DAISY CHAINING

- It is a centralized bus arbitration method. During any bus cycle, the bus master may be any device – the processor or any DMA controller unit, connected to the bus.
- Bus control passes from one bus master to the next one, then to the next and so on. That is from controller units C0 to C1, then to C2, then U3, and so on.



Daisy Chaining

Sequence of Signals in the arbitration process

- Bus-grant signal (BG) which functions like a token is first sent to C0.
- If C0 does not need the bus, it passes BG to C1.
- A controller needing the bus raises a bus request (BR) signal.
- A bus-busy (BUSY) signal generates when that controller becomes the bus master.

Signals in the arbitration process

- When bus master no longer needs the bus, it deactivates BR and BUSY signal also deactivates.
- Another BG is issued and passed from C0 to down the priority controllers one by one
- [For example, COM2 to COM1 in IBM PC]

Advantages

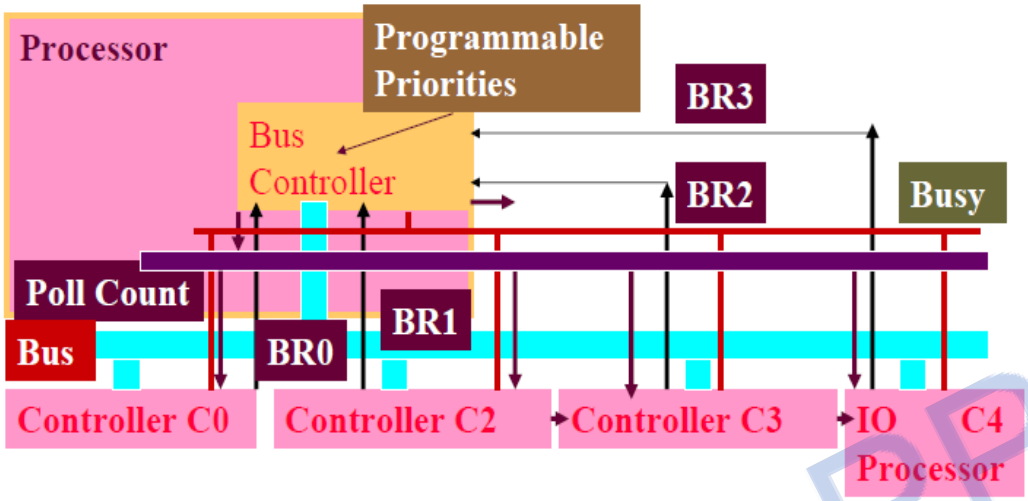
- Simplicity and Scalability.
- The user can add more devices anywhere along the chain, up to a certain maximum value.

Disadvantages

- The value of priority assigned to a device is depends on the position of master bus.

- Propagation delay is arises in this method.
- If one device fails then entire system will stop working.

POLLING OR ROTATING PRIORITY METHOD



Diag: Polling method for Bus Sharing by Multiple Processors or controllers

- BUSY activates when that controller becomes the bus master. When BR deactivates, then BG and BUSY also deactivates and counts increment starts.
- Polling method advantage is that the controller next to the current bus master gets the highest priority to the access the bus after the current bus master finishes the operations through the bus.
- A poll counts value is sent to the controllers and is incremented. Assume that there are 8 controllers. Three poll count signals p2, p1, p0 successively change from 000, 001, …, 110, 111, 000, … If on count = i, a BR signal is received then counts increment stops, BG is sent.

Advantages:

- This method does not favor any particular device and processor.
- The method is also quite simple.
- If one device fails then entire system will not stop working.

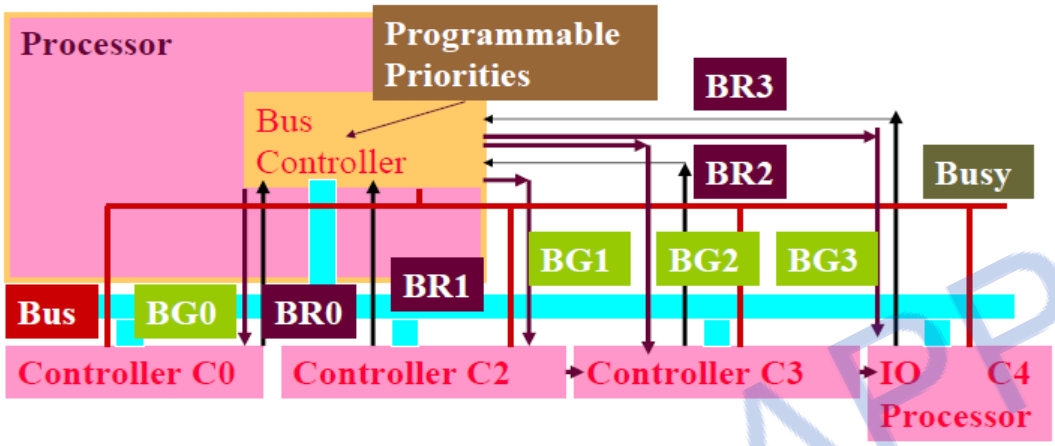
Disadvantages:

- Adding bus masters is different as increases the number of address lines of the circuit.

FIXED PRIORITY or INDEPENDENT REQUEST AND GRANT METHOD

Controller separate BR signals, BR0, BR1,…, BRn.

- Separate BG signals, BG0, BG1, …, BGn for the controllers.



Diag: Independent bus request method

- An ith controller sends BRi (i-th bus request signal) and when it receives BGi (ith bus grant signal), it uses the bus and then BUSY signal activates.
- Any controller, which finds active BUSY, does not send BR from it.

Advantages

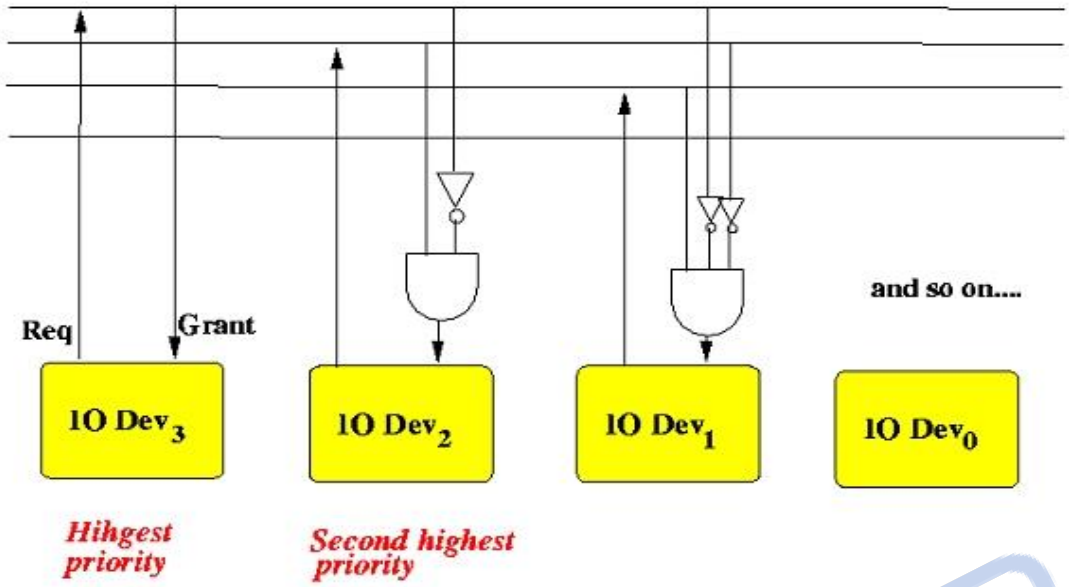
- This method generates fast response.

Disadvantages

- Hardware cost is high as large number of control lines is required.

DISTRIBUTED BUS ARBITRATION:

Here, all the devices participate in the selection of the next bus master. Each device on the bus is assigned a4 bit identification number. The priority of the device will be determined by the generated ID.



- When one or more devices request control of the bus, they assert the start arbitration signal and place their 4-bit identification numbers on arbitration lines through ARB0 to ARB3.
- Each device compares the code and changes its bit position accordingly. It does so by placing a 0 at the input of their drive.
- The distributed arbitration is highly reliable because the bus operations are not dependent on devices.

INTERFACE CIRCUITS

An I/O interface consists of the circuitry required to connect an I/O device to a computer bus. On one side of the interface, we have bus signals. On the other side, we have a data path with its associated controls to transfer data between the interface and the I/O device – port. We have two types:

- **Parallel port**
- **Serial port**

A parallel port transfers data in the form of a number of bits (8 or 16) simultaneously to or from the device. A serial port transmits and receives data one bit at a time. Communication with the bus is the same for both formats. The

conversion from the parallel to the serial format, and vice versa, takes place inside the interface circuit. In parallel port, the connection between the device and the computer uses a multiple-pin connector and a cable with as many wires. This arrangement is suitable for devices that are physically close to the computer. In serial port, it is much more convenient and cost-effective where longer cables are needed.

Typically, the functions of an I/O interface are:

• Provides a storage buffer for at least one word of data

• Contains status flags that can be accessed by the processor to determine whether the buffer is full or empty

• Contains address-decoding circuitry to determine when it is being addressed by the processor

• Generates the appropriate timing signals required by the bus control scheme

• Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

Parallel Port

Input Port

Example1: Keyboard to Processor

Observe the parallel input port that connects the keyboard to the processor. Now, whenever the key is tapped on the keyboard an electrical connection is established that generates an electrical signal. This signal is encoded by the encoder to convert it into ASCII code for the corresponding character pressed at the keyboard (as shown in below figure)

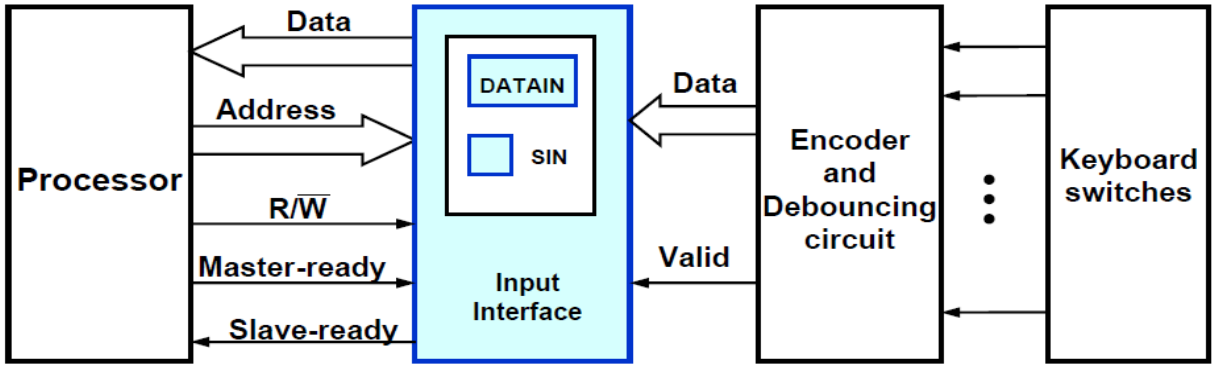


Fig: Parallel Input Port that connect Keyboard and Processor

The Hardware components needed are

– Status flag, SIN
– R/~W
– Master-ready
– Address decoder

When a key is pressed, the valid signal changes from 0 to 1 causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register

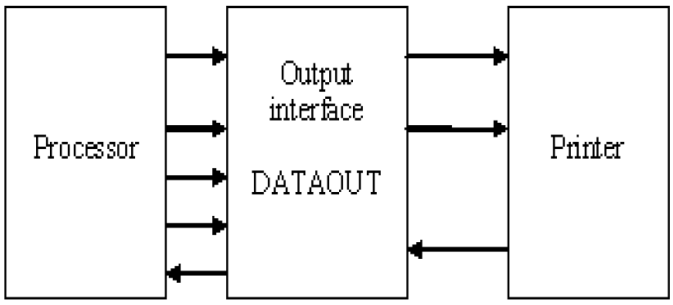
Example 2: Printer to processor

The hardware components needed for connecting a printer to a processor are:

The circuit of output interface, and

– Slave-ready

– R/W
– Master-ready
– Address decoder
– Handshake control

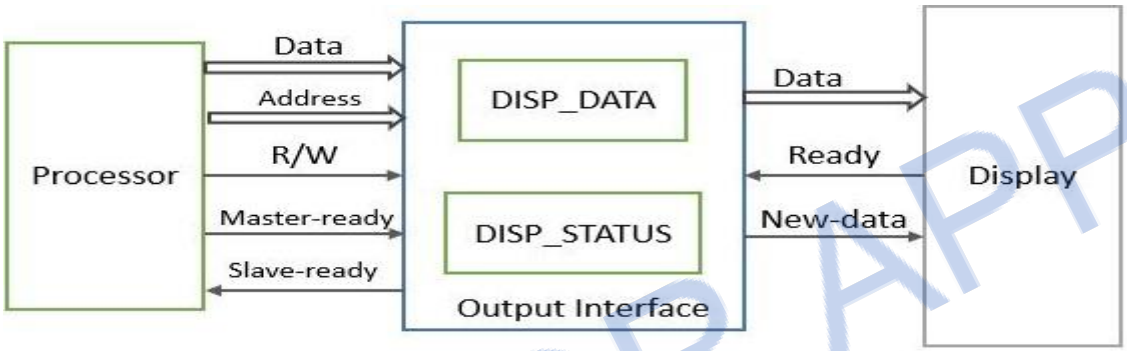


Printer to processor connection

The input and output interfaces can be combined into a single interface. The general purpose parallel interface circuit can be configured in a variety of ways. For increased flexibility, the circuit makes it possible for some lines to serve as inputs and some lines to serve as outputs, under program control.

Output Port

When the display unit is ready to display a character, it activates its ready line to 1 which setups the DOUT flag in the DISP_STATUS register to 1. This indicates the processor and the processor places the character to the DISP_DATA register.

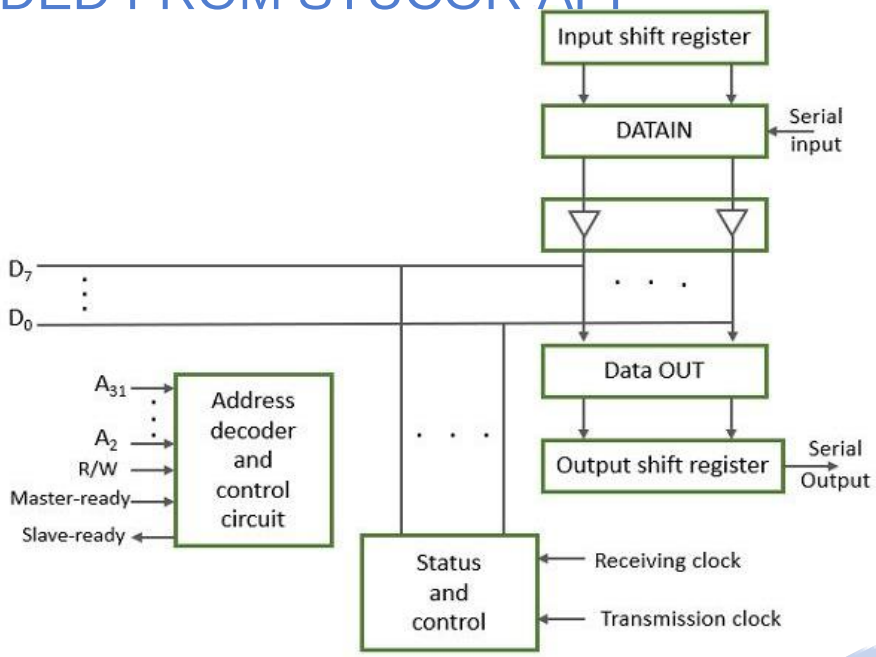


Parallel Output Port that connect Display and Processor

As soon as the processor loads the character in the DISP_DATA the DOUT flag setbacks to 0 and the New-data line to 1. Now as the display senses that the new-data line is activated it turns the ready line to 0 and accepts the character in the DISP_DATA register to display it.

Serial Port

Opposite to the parallel port, the serial port connects the processor to devices that transmit only one bit at a time. Here on the device side, the data is transferred in the bit-serial pattern, and on the processor side, the data is transferred in the bit-parallel pattern.



Serial Interface Circuit

The transformation of the format from serial to parallel i.e., from device to processor, and from parallel to serial i.e., from processor to device is made possible with the help of shift registers (input shift register & output shift register).

The input shift register accepts the one bit at a time in a bit-serial fashion till it receives all 8 bits. When all the 8 bits are received by the input shift register it loads its content into the DATA IN register parallelly. In a similar fashion, the content of the DATA OUT register is transferred in parallel to the output shift register as shown in the given figure

The serial interface port connected to the processor via system bus functions similarly to the parallel port. The status and control block has two status flags SIN and SOUT. The SIN flag is set to 1 when the I/O device inputs the data into the DATA IN register through the input shift register and the SIN flag is cleared to 0 when the processor reads the data from the DATA IN register.

When the value of the SOUT register is 1 it indicates to the processor that the DATA OUT register is available to receive new data from the processor. The processor writes the data into the DATA OUT register and sets the SOUT flag to 0

and when the output shift register reads the data from the DATA OUT register sets back SOUT to 1.

There are two techniques to transmit data using the encoding scheme.

1. Asynchronous Serial Transmission
2. Synchronous Serial Transmission

Asynchronous Serial Transmission

- In the asynchronous transmission, the clock used by the transmitter and receiver is not synchronized. So, the bits to be transmitted are grouped into a group of 6 to 8 bits which has a defined starting bit and ending bit. The start bit has a logic value 0 and the stop bit has a logic value 1.
- The data received at the receiver end is recognized by this start and stop bit. This approach is useful where is the transmission is slow.

Synchronous Serial Transmission

- The start and stop bit we used in the asynchronous transmission provides the correct timing information but this approach is not useful where the transmission speed is high.
- So, in the synchronous transmission, the receiver generates the clock that is synchronized with the clock of the transmitter. This lets the transmitting large blocks of data at a high speed.

Standard I/O interfaces

Consider a computer system using different interface standards. Let us look in to Processor bus and Peripheral Component Interconnect (PCI) bus. These two buses are interconnected by a circuit called bridge. It is a bridge between processor bus and PCI bus. An example of a computer system using different interface standards is shown in figure 4.38. The three major standard I/O interfaces discussed here are:

– PCI (Peripheral Component Interconnect)

– SCSI (Small Computer System Interface)

– USB (Universal Serial Bus)

PCI (Peripheral Component Interconnect)

Host, main memory and PCI bridge are connected to disk, printer and Ethernet interface through PCI bus. At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an initiator in PCI terminology. This is either processor or DMA controller. The addressed device that responds to read and write commands is called a target. A complete transfer operation on the bus, involving an address and a burst of data, is called a transaction. Device configuration is also discussed.

SCSI Bus

It is a standard bus defined by the American National Standards Institute (ANSI). A controller connected to a SCSI bus is an initiator or a target. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

• The SCSI controller contends for control of the bus (initiator).

• When the initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.

• The target starts an output operation. The initiator sends a command specifying the required read operation.

• The target sends a message to the initiator indicating that it will temporarily suspends the connection between them. Then it releases the bus. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation.

• The target transfers the contents of the data buffer to the initiator and then suspends the connection again.

• The target controller sends a command to the disk drive to perform another seek operation.

• As the initiator controller receives the data, it stores them into the main memory using the DMA approach.

• The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

USB (UNIVERSAL SERIAL BUS)

Universal Serial Bus, **USB** is a plug and play interface that allows a computer to communicate with peripheral and other devices. USB-connected devices cover a broad range; anything from keyboards and mice, to music players and flash drives.

The USB has been designed to meet several key objectives

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections
- Enhance user convenience through a "plug-and-play" mode of operation

USB devices

Today, there are millions of different USB devices that can be connected to your computer. The list below contains just a few of the most common.

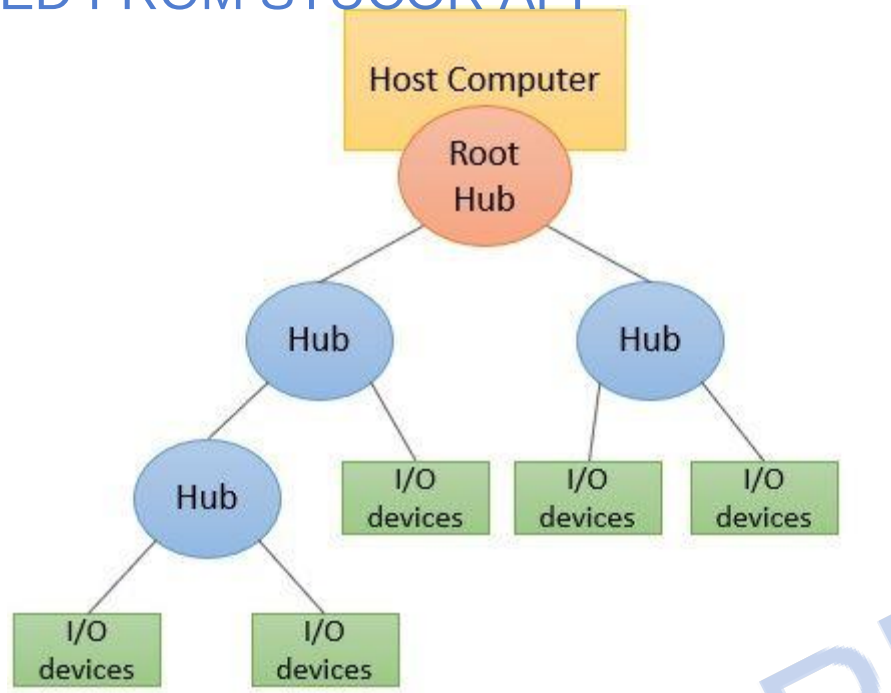
- Digital Camera
- External drive
- iPod or other MP3 players
- Keyboard
- Keypad

- Microphone
- Mouse
- Printer
- Joystick
- Scanner
- Smartphone
- Tablet
- Webcams

USB Architecture

When multiple I/O devices are connected to the computer through USB they all are organized in a tree structure. Each I/O device makes a *point-to-point* connection and transfers data using the *serial transmission format* we have discussed serial transmission in our previous content 'interface circuit'.

As we know a tree structure has a **root, nodes** and **leaves.** The tree structure connecting I/O devices to the computer using USB has nodes which are also referred to as a **hub**. Hub is the intermediatory connecting point between the I/O devices and the computer. Every tree has a root here, it is referred to as the **root hub** which connects the entire tree to the hosting computer. The leaves of the tree here are nothing but the I/O devices such as a mouse, keyboard, camera, speaker.



Universal Serial Bus Tree Structure

USB Protocols

All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information

- The information transferred on the USB can be divided into two broad categories: control and data
- Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error
- Data packets carry information that is delivered to a device. For example, input and output data are transferred inside data packets

USB Device States

A USB device can have several possible states as described below:

• **Attached State:** This state occurs when the device is attached to the Host.

• **Powered State:** After the device is attached, the Host provides power to the device if it does not have its own power supply. The device should not draw more than 100 mA in this state.

• **Default State:** This state occurs when the device is reset and has not been assigned a unique address. In this state the device uses default control pipe for communication and default address 0.

• **Addressed State:** The USB device enters this state after it gets a unique address which is used for future communications.

• **Configured:** When the Host obtains required information from the device, it loads the appropriate driver for the device. The host configures the device by selecting a configuration. The device is now ready to do the operations it was meant for.

• **Suspended State:** The USB device enters the suspended state when the bus remains idle for more than 3mS. In this state, the device must not draw more than 500uA of current.

****