

Lecture 1

Introduction, Concepts and laws of Boolean algebra

Introduction

A digital computer stores data in terms of digits (numbers) and proceeds in discrete steps from one state to the next. The states of a digital computer typically involve binary digits which may take the form of the presence or absence of magnetic markers in a storage medium, on-off switches or relays. In digital computers, even letters, words and whole texts are represented digitally.

Digital Logic is the basis of electronic systems, such as computers and cell phones. Digital Logic is rooted in binary code, a series of zeroes and ones each having an opposite value. This system facilitates the design of electronic circuits that convey information, including logic gates. Digital Logic gate functions include and, or and not. The value system translates input signals into specific output. Digital Logic facilitates computing, robotics and other electronic applications.

Digital Logic Design is foundational to the fields of electrical engineering and computer engineering. Digital Logic designers build complex electronic components that use both electrical and computational characteristics. These characteristics may involve power, current, logical function, protocol and user input. Digital Logic Design is used to develop hardware, such as circuit boards and microchip processors. This hardware processes user input, system protocol and other data in computers, navigational systems, cell phones or other high-tech systems.

Boolean Algebra

One of the primary requirements when dealing with digital circuits is to find ways to make them as simple as possible. This constantly requires that complex logical expressions be reduced to simpler expressions that nevertheless produce the same results under all possible conditions. The simpler expression can then be implemented with a smaller, simpler circuit, which in turn saves the price of the unnecessary gates, reduces the number of gates needed, and reduces the power and the amount of space required by those gates. One tool to reduce logical expressions is the mathematics of logical expressions, introduced by George Boole in 1854 and known today as *Boolean Algebra*. The rules of Boolean Algebra are simple and straight-forward, and can be applied to any logical expression. The resulting reduced expression can then be readily tested with a Truth Table, to verify that the reduction was valid. Boolean algebra is an algebraic structure defined on a set of elements B , together with two binary operators (+, \cdot) provided the following postulates are satisfied.

1. Closure with respect to operator + and Closure with respect to operator \cdot

2. An identity element with respect to + designated by 0: $X+0= 0+X=X$

An identity element with respect to \cdot designated by 1: $X \cdot 1= 1 \cdot X=X$

3. Commutative with respect to +: $X+Y=Y+X$

Commutative with respect to \cdot : $X \cdot Y=Y \cdot X$ 4. .

distributive over +: $X \cdot (Y+Z)=X \cdot Y+X \cdot Z$

+ distributive over \cdot : $X+(Y \cdot Z)=(X+Y) \cdot (X+Z)$

5. For every element x belonging to B, there exist an element x' or called the complement of x such that $x \cdot x'=0$ and $x+x'=1$

6. There exists at least two elements x,y belonging to B such that $x \neq y$

The two valued Boolean algebra is defined on a set $B=\{0,1\}$ with two binary operators + and \cdot .

X	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	0

x	x'
0	1
1	0

Closure. from the tables, the result of each operation is either 0 or 1 and 1,0 belongs to B

Identity. From the truth table we see that 0 is the identity element for + and 1 is the identity element for \cdot .

Commutative law is obvious from the symmetry of binary operators table.

Distributive Law. $x \cdot (y+z)=x \cdot y+x \cdot z$

x	y	z	$y+z$	$x \cdot (y+z)$	$x \cdot y$	$x \cdot z$	$x \cdot y+x \cdot z$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Distributive of + over \cdot can be shown as in the truth table above

From the complement table we can see that $x+x'=1$ i.e $1+0=1$ and $x \cdot x'=0$ i.e $1 \cdot 0=0$

Principle of duality of Boolean algebra

The principle of duality state that every algebraic expression which can be deduced from the postulates of Boolean algebra remains valid if the operators and the identity elements are interchanged. This mean

that the dual of an expression is obtained changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa

Laws of Boolean algebra

Postulate 2:

$$(a) 0 + A = A$$

$$(b) 1.A = A$$

Postulate 5 :

$$(a) A + A' = 1$$

$$(b) A . A' = 0$$

Theorem1 : Identity Law

$$(a) A + A = A$$

$$(b) A . A = A$$

Theorem2

$$(a) 1 + A = 1$$

$$(b) 0 . A = 0$$

Theorem3: involution

$$A'' = A$$

Postulate 3: Commutative Law

$$(a) A + B = B + A$$

$$(b) A . B = B . A$$

Theorem4: Associate Law

$$(a) (A + B) + C = A + (B + C) \quad (b) (A . B) . C = A . (B . C)$$

Postulate4: Distributive Law

$$(a) A . (B + C) = A . B + A . C$$

$$(b) A + (B . C) = (A + B) . (A + C)$$

Lecture 2

Boolean functions and Representation in SOP and POS forms

Boolean functions

Operations of binary variables can be described by mean of appropriate mathematical function called Boolean function. A Boolean function define a mapping from a set of binary input values into a set of output values. A Boolean function is formed with binary variables, the binary operators AND and OR and the unary operator NOT.

For example , a Boolean function $f(x_1, x_2, x_3, \dots, x_n) = y$ defines a mapping from an arbitrary combination of binary input values $(x_1, x_2, x_3, \dots, x_n)$ into a binary value y . a binary function with n input variable can operate on 2^n distincts values. Any such function can be described by using a truth table consisting of 2^n rows and n columns. The content of this table are the values produced by that function when applied to all the possible combination of the n input variable.

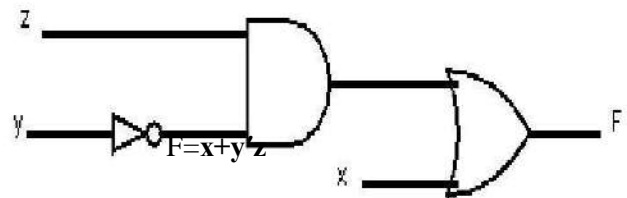
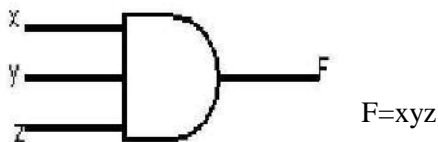
Example

x	y	x.y
0	0	0
0	1	0
1	0	0
1	1	1

The function f , representing $x.y$, that is $f(x,y)=xy$. Which mean that $f=1$ if $x=1$ and $y=1$ and $f=0$ otherwise.

For each rows of the table, there is a value of the function equal to 1 or 0. The function f is equal to the sum of all rows that gives a value of 1.

A Boolean function may be transformed from an algebraic expression into a logic diagram composed of AND, OR and NOT gate. When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate and each term is implemented with a logic gate . e.g.



Standard form

Another way to express a boolean function is in standard form. Here the term that forms the function may contain one, two or any number of literals. There are two types of standard form. The **sum of product (SOP)** and the **product of sum (POS)**.

The sum of product (SOP) is a Boolean expression containing AND terms called product terms of one or more literals each. The sum denotes the ORing of these terms

e.g. $F = x + xy' + x'yz$

The product of sum (POS) is a Boolean expression containing OR terms called SUM terms. Each term may have any number of literals. The product denotes the ANDing of these terms

e.g. $F = (x + y')(x' + y + z)$

A boolean function may also be expressed in a non standard form. In that case, distributive law can be used to remove the parenthesis

$$F = (xy + zw)(x'y' + z'w')$$

$$= xy(x'y' + z'w') + zw(x'y' + z'w')$$

$$= xyx'y' + xyz'w' + zwx'y' + zwz'w'$$

$$= xyz'w' + zwx'y'$$

A Boolean equation can be reduced to a minimal number of literals by algebraic manipulation. Unfortunately, there are no specific rules to follow that will guarantee the final answer. The only method is to use the theorem and postulate of Boolean algebra and any other manipulation that becomes familiar.

Describing existing circuits using Logic expressions

To define what a combinatorial circuit does, we can use a *logic expression* or an *expression* for short. Such an expression uses the two constants 0 and 1, variables such as x , y , and z (sometimes with suffixes) as names of inputs and outputs, and the operators $+$, \cdot and a horizontal bar or a prime (which stands for *not*). As usual, multiplication is considered to have higher priority than addition. Parentheses are used to modify the priority.

If Boolean functions in either Sum of Product or Product of Sum forms can be implemented using 2-

Level implementations.

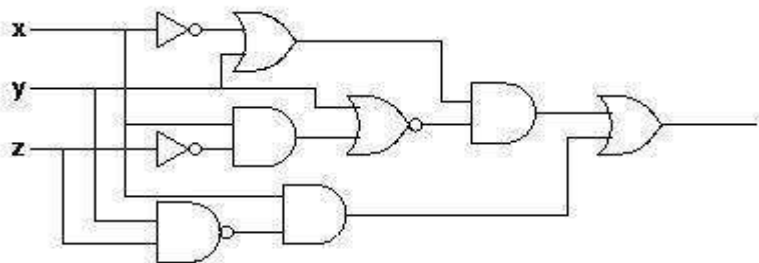
For SOP forms AND gates will be in the first level and a single OR gate will be in the second level.

For POS forms OR gates will be in the first level and a single AND gate will be in the second level.

Examples:

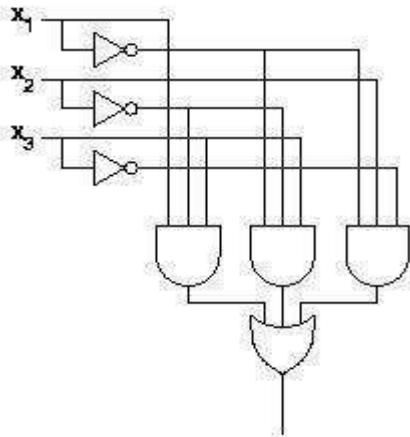
$$(X'+Y)(Y+XZ)'+X(YZ)'$$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



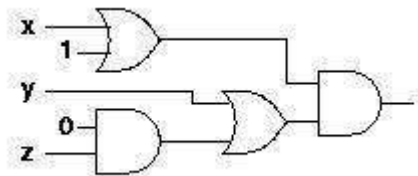
$$X_1X_2'X_3+X_1'X_2'X_2+X_1'X_2X_3$$

The equation is in sum of product. The implementation is in 2-Levels. AND gates form the first level and a single OR gate the second level.



$$(X+1)(Y+0Z)$$

The equation is neither in sum of product nor in product of sum. The implementation is as follow



Lecture 3

Minterm and maxterm , Minimization of logic expressions by Karnaugh Map

Canonical form (Minterms and Maxterms)

A binary variable may appear either in its normal form or in its complement form. Consider two binary variables x and y combined with AND operation. Since each variable may appear in either form **there are four possible combinations: $x'y'$, $x'y$, xy' , xy** . Each of the terms represents one distinct area in the Venn diagram and is called minterm or a standard product. With n variables, 2^n minterms can be formed.

In a similar fashion, n variables forming an OR term provide 2^n possible combinations called maxterms or standard sum. Each maxterm is obtained from an OR term of the n variables, with each variable being primed if the corresponding bit is 1 and un-primed if the corresponding bit is 0. Note that each maxterm is the complement of its corresponding minterm and vice versa.

X	Y	Z	Minterm	maxterm
0	0	0	$x'y'z'$	$X+y+z$
0	0	1	$x'y'z$	$X+y+z'$
0	1	0	$x'yz'$	$X+y'+z$
0	1	1	$x'yz$	$X+y'+z'$
1	0	0	$xy'z'$	$X'+y+z$
1	0	1	$xy'z$	$X'+y+z'$
1	1	0	xyz'	$X'+y'+z$
1	1	1	xyz	$X'+y'+z'$

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of variables that produce a 1 and taking the OR of those terms.

Similarly, the same function can be obtained by forming the maxterm for each combination of variables that produces 0 and then taking the AND of those terms.

It is sometimes convenient to express the Boolean function when it is in sum of minterms, in the following notation:

$F(X,Y,Z) = \sum(1,4,5,6,7)$. The summation symbol \sum stands for the ORing of the terms; the numbers following it are the minterms of the function. The letters in the parenthesis following F form a list of the variables in the order taken when the minterm is converted to an AND term.

$$\text{So, } F(X,Y,Z) = \sum(1,4,5,6,7) = X'Y'Z + XY'Z' + XY'Z + XYZ' + XYZ$$

Sometimes it is convenient to express a Boolean function in its sum of minterms. If it is not in that case, the expression is expanded into the sum of AND terms and if there is any missing variable, it is **ANDed with an expression such as $x+x'$ where x is one of the missing variables**.

To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This can be done by using distributive law $x+xz=(x+y)(x+z)$. then if there is any missing variable, say x in each OR term is **ORded with x'** .

e.g. represent $F=xy+x'z$ as a product of maxterm

$$=(xy +x')(xy+z)$$

$$(x+x')(y+x')(x+z)(y+z)$$

$$(y+x')(x+z)(y+z)$$

Adding missing variable in each term

$$(y+x')= x'+y+zz' \quad = (x'+y+z)(x'+y+z')$$

$$(x+z)= x+z+yy' \quad = (x+y+z)(x+y'+z)$$

$$(y+z)= y+z+xx' = (x+y+z)(x'+y+z)$$

$$F = (x+y+z)(x+y'+z)(x'+y+z)(x'+y+z')$$

$$x'+y+z'$$

A convenient way to express this function is as follow :

$$F(x,y,z) = \prod (0,2,4,5)$$

Karnaugh map

The Karnaugh map also known as Veitch diagram or simply as K map is a two dimensional form of the truth table, drawn in such a way that the simplification of Boolean expression can be immediately **be seen from the location of 1's in the map. The map is a diagram made up of squares , each square** represent one minterm. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognised graphically in the map from the area enclosed by those squares whose minterms are included in the function.

A two variable Boolean function can be represented as follow

		B	
		0	1
A	0	$A'B'$ 0	$A'B$ 1
	1	AB' 2	AB 3

A three variable function can be represented as follow



		yz			
x		00	01	11	10
0		$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
1		$xy'z'$	$xy'z$	xyz	xyz'

A four variable Boolean function can be represented in the map bellow

		AB					
		00	01	11	10	ABCD	ABCD
CD	00	0	4	12	8	0000 - 0	1000 - 8
	01	1	5	13	9	0001 - 1	1001 - 9
	11	3	7	15	11	0010 - 2	1010 - 10
	10	2	6	14	10	0011 - 3	1011 - 11
						0100 - 4	1100 - 12
						0101 - 5	1101 - 13
						0110 - 6	1110 - 14
						0111 - 7	1111 - 15

To simplify a Boolean function using karnaugh map, the first step is to plot all ones in the **function truth table on the map**. The next step is to combine adjacent 1's into a group of one, two, four, eight, sixteen. The group of minterm should be as large as possible. A single group of four minterm yields a simpler expression than two groups of two minterms.

In a four variable karnaugh map,

1 variable product term is obtained if 8 adjacent squares are covered

2 variable product term is obtained if 4 adjacent squares are covered

3 variable product term is obtained if 2 adjacent squares are covered

1 variable product term is obtained if 1 square is covered

A square having a 1 may belong to more than one term in the sum of product expression

The final stage is reached when each of the group of minterms are ORed together to form the simplified sum of product expression

The karnaugh map is not a square or rectangle as it may appear in the diagram. The top edge is adjacent to the bottom edge and the left hand edge adjacent to the right hand edge. Consequent, two squares in karnaugh map are said to be adjacent if they differ by only one variable

Minimization of Boolean expressions using Karnaugh maps.

Given the following truth table for the majority function.

a	b	c	M(output)
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

The Boolean algebraic expression is

$$m = a'bc + ab'c + abc' + abc.$$

the minimization using algebraic manipulation can be done as follows.

$$m = a'bc + abc + ab'c + abc + abc' + abc$$

$$= (a' + a)bc + a(b' + b)c + ab(c' + c)$$

$$= bc + ac + ab$$

The **abc** term was replicated and combined with the other terms.

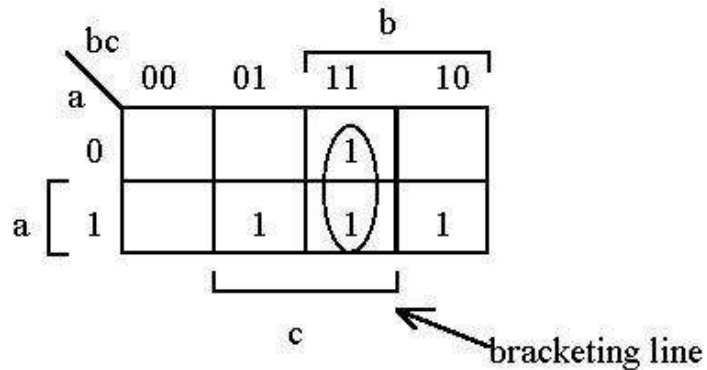
To use a Karnaugh map we draw the following map which has a position (square) corresponding to each of the 8 possible combinations of the 3 Boolean variables. The upper left position corresponds to the 000 row of the truth table, the lower right position corresponds to 110. Each square has two coordinates, the vertical coordinate corresponds to the value of variable **a** and the horizontal corresponds to the values of **b** and **c**.

		bc			
		00	01	11	10
a	0			1	
	1		1	1	1

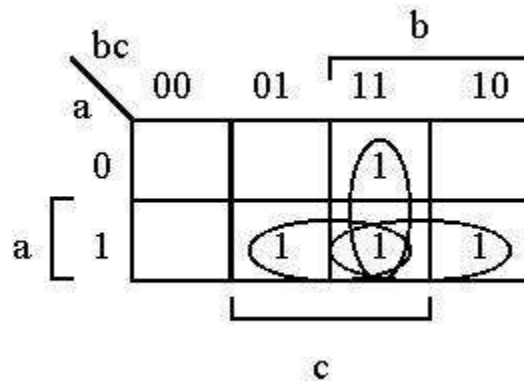
c

The 1s are in the same places as they were in the original truth table. The 1 in the first row is at position 110 (**a** = 1, **b** = 1, **c** = 0). The minimization is done by drawing circles around sets of adjacent 1s.

Adjacency is horizontal, vertical, or both. The circles must always contain 2^n 1s where n is an integer.



We have circled two 1s. The fact that the circle spans the two possible values of a (0 and 1) means that the a term is eliminated from the Boolean expression corresponding to this circle. The bracketing lines shown above correspond to the positions on the map for which the given variable has the value 1. The bracket delimits the set of squares for which the variable has the value 1. We see that the two circled 1s are at the intersection of sets b and c , this means that the Boolean expression for this set of bc .



Now we have drawn circles around all the 1s. The left bottom circle is the term ac . Note that the circle spans the two possible values of b , thus eliminating the b term. Another way to think of it is that the set of squares in the circle contains the same squares as the set a intersected with the set c . The other circle (lower right) corresponds to the term ab . Thus the expression reduces to

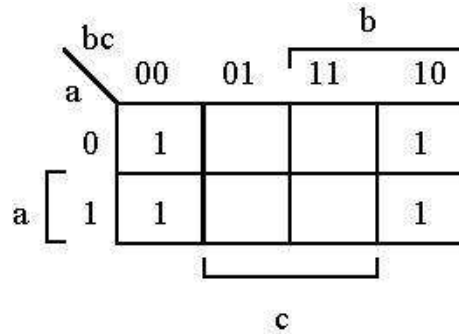
$bc + ac + ab$ as we saw before.

What is happening? What does adjacency and grouping the 1s together have to do with minimization? Notice that the 1 at position 111 was used by all 3 circles. This 1 corresponds to the abc term that was replicated in the original algebraic minimization. Adjacency of 2 1s means that the terms corresponding to those 1s differ in one variable only. In one case that variable is negated and in the other it is not.

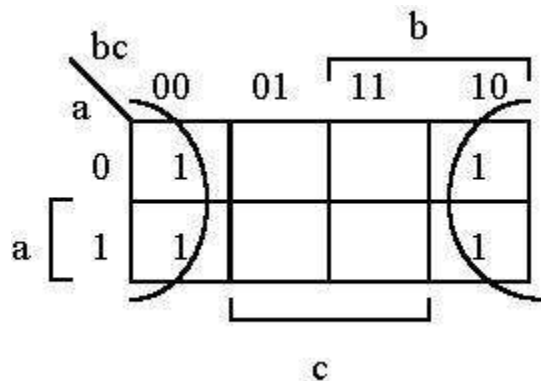
For example, in the first map above, the one with only 1 circle. The upper 1 is the term $a'bc$ and the lower is abc . Obviously they combine to form bc ($a'bc + abc = (a' + a)bc = bc$). That is exactly what we got using the map.

The map is easier than algebraic minimization because we just have to recognize patterns of 1s in the map instead of using the algebraic manipulations. Adjacency also applies to the edges of the map.

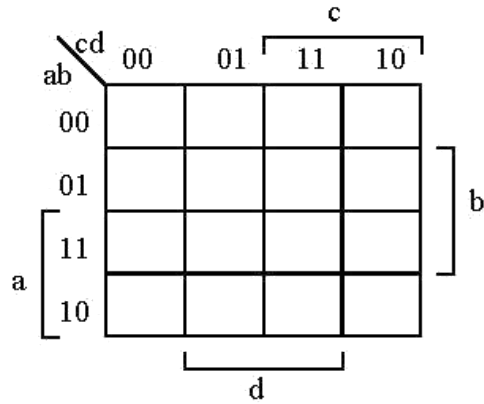
Let's try another 3 variable map.



At first it may seem that we have two sets, one on the left of the map and the other on the right. Actually there is only 1 set because the left and right are adjacent as are the top and bottom. The expression for all 4 1s is c' . Notice that the 4 1s span both values of a (0 and 1) and both values of b (0 and 1). Thus, only the c value is left. The variable c is 0 for all the 1s, thus we have c' . The other way to look at it is that the 1's overlap the horizontal b line and the short vertical a line, but they all lay outside the horizontal c line, so they correspond to c' . (The horizontal c line delimits the c set. The c' set consists of all squares outside the c set. Since the circle includes all the squares in c' , they are defined by c' . Again, notice that both values of a and b are spanned, thus eliminating those terms.)

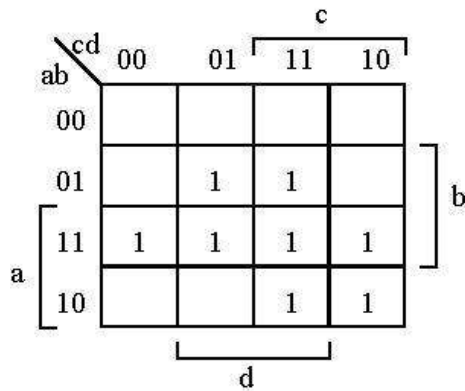


Now for 4 Boolean variables. The Karnaugh map is drawn as shown below.



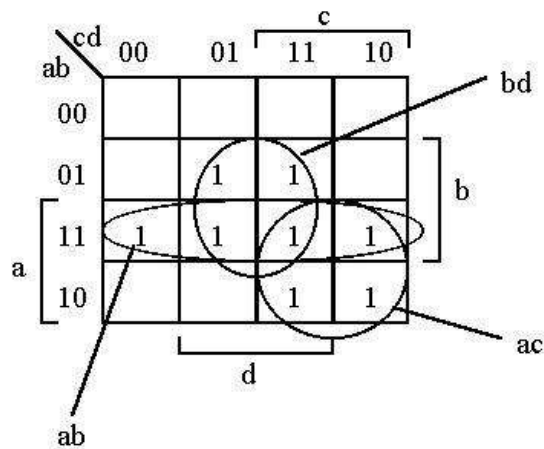
The following corresponds to the Boolean expression

$$q = a'bc'd + a'bcd + abc'd' + abc'd + abcd + abcd' + ab'cd + ab'cd'$$



RULE: Minimization is achieved by drawing the smallest possible number of circles, each containing the largest possible number of 1s.

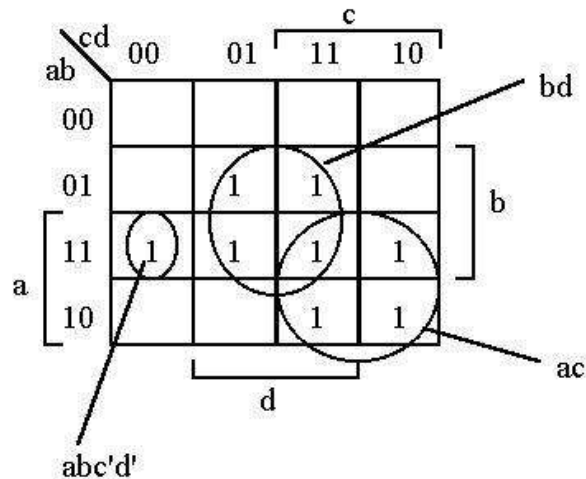
Grouping the 1s together results in the following.



The expression for the groupings above is

$$q = bd + ac + ab$$

This expression requires 3 2-input **and** gates and 1 3-input **or** gate. We could have accounted for all the 1s in the map as shown below, but that results in a more complex expression requiring a more complex gate.



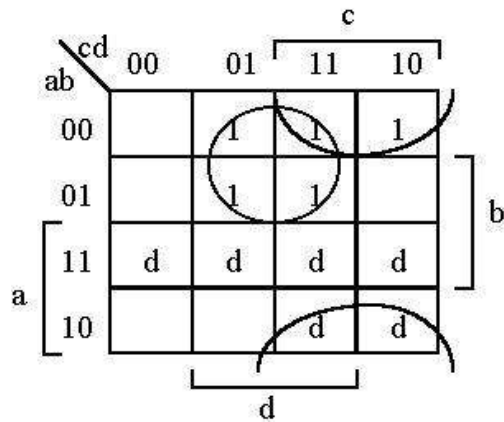
The expression for the above is $bd + ac + abc'd'$. This requires 2 2-input **and** gates, a 4-input **and** gate, and a 3 input **or** gate. Thus, one of the **and** gates is more complex (has two additional inputs) than required above. Two inverters are also needed.

Don't Cares

Sometimes we do not care whether a 1 or 0 occurs for a certain set of inputs. It may be that those inputs will never occur so it makes no difference what the output is. For example, we might have a bcd (binary coded decimal) code which consists of 4 bits to encode the digits 0 (0000) through 9 (1001). The remaining codes (1010 through 1111) are not used. If we had a truth table for the prime numbers 0 through 9, it would be

abcd	p
0000	0
0001	1
0010	1
0011	1
0100	0
0101	1
0110	0
0111	1
1000	0
1001	0
1010	d
1011	d
1100	d
1101	d
1110	d
1111	d

The ds in the above stand for "don't care", we don't care whether a 1 or 0 is the value for that combination of inputs because (in this case) the inputs will never occur.



The circle made entirely of 1s corresponds to the expression $a'd$ and the combined 1 and d circle (actually a combination of arcs) is $b'c$. Thus, if the disallowed input 1011 did occur, the output would be 1 but if the disallowed input 1100 occurs, its output would be 0. The minimized expression is

$$p = a'd + b'c$$

Notice that if we had ignored the ds and only made a circle around the 2 1s, the resulting expression would have been more complex, $a'b'c$ instead of $b'c$.

Lecture 4 & Lecture 5

Adder and Subtractor (half-full adder & subtractor)

Adders

In electronics, an adder or summer is a digital circuit that performs addition of numbers. In modern computers adders reside in the arithmetic logic unit (ALU) where other operations are performed. Although adders can be constructed for many numerical representations, such as Binary-coded decimal or excess-3, the most common adders operate on binary numbers. In cases where two's complement or ones complement is being used to represent negative numbers, it is trivial to modify an adder into an adder-subtractor. Other signed number representations require a more complex adder.

-Half Adder

A half adder is a logical circuit that performs an addition operation on two binary digits. The half adder produces a sum and a carry value which are both binary digits.

A half adder has two inputs, generally labelled A and B, and two outputs, the sum S and carry C. S is the two-bit XOR of A and B, and C is the AND of A and B. Essentially the output of a half adder is the sum of two one-bit numbers, with C being the most significant of these two outputs.

The drawback of this circuit is that in case of a multibit addition, it cannot include a carry.

Following is the truth table for a half adder:

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

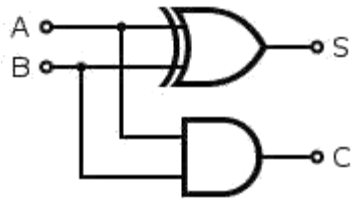
Equation of the Sum and Carry.

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

$$\text{Carry} = AB$$

One can see that Sum can also be implemented using XOR gate

(a) Truth Table of Half Adder



(b) Logic Diagram of Half Adder

Figure 1. (a) Truth Table and (b) Logic Diagram of Half Adder

Full Adder.

A full adder has three inputs A , B , and a carry in C_i , such that multiple adders can be used to add larger numbers. To remove ambiguity between the input and output carry lines, the carry in is labelled C_i or C_{in} while the carry out is labelled C_o or C_{out} .

A full adder is a logical circuit that performs an addition operation on three binary digits. The full adder produces a sum and carry value, which are both binary digits. It can be combined with other full adders or work on its own.

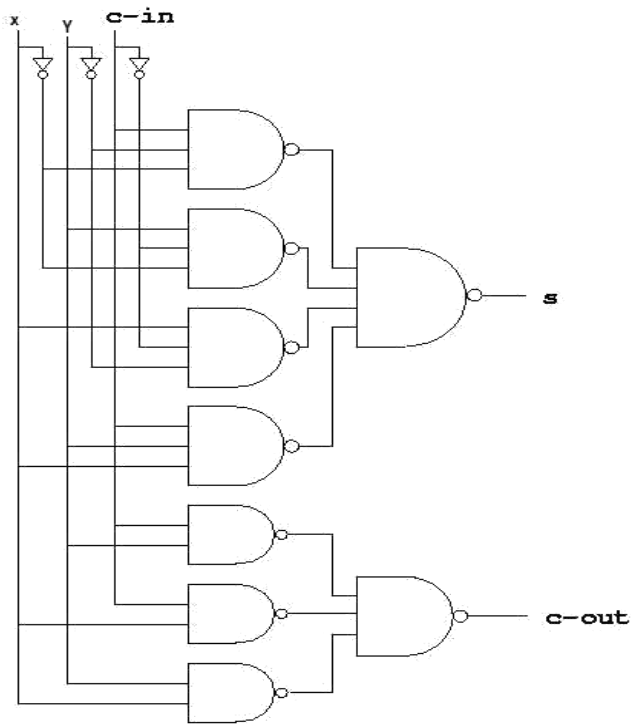
Input			Output	
A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table of full adder

$$C_o = A'B C_i + A B' C_i + A B C_i' + A B C_i$$

$$S = A'B'C_i + A'BC_i' + ABC_i' + ABC_i$$

A full adder can be trivially built using our ordinary design methods for combinatorial circuits. Here is the resulting circuit diagram using NAND gates only:



(b) Logic Diagram Of full adder

$$C_o = A'BC_i + AB'C_i + ABC_i' + ABC_i$$

By manipulating C_o , we can see that $C_o = C_i(A \oplus B) + AB$

$$S = A'B'C_i + A'BC_i' + ABC_i' + ABC_i$$

By manipulating S , we can see that $S = C_i \oplus (A \oplus B)$

Note that the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. This is because the only discrepancy between OR and XOR gates occurs when both inputs are 1; for the adder shown here, this is never possible. Using only two types of gates is convenient if one desires to implement the adder directly using common IC chips.

A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, connecting the sum from that to an input to the second adder, connecting C_i to the other input and OR the two carry outputs. Equivalently, S could be made the three-bit xor of A, B, and C_i and C_o could be made the three-bit majority function of A, B, and C_i . The output of the full adder is the two-bit arithmetic sum of three one-bit numbers.

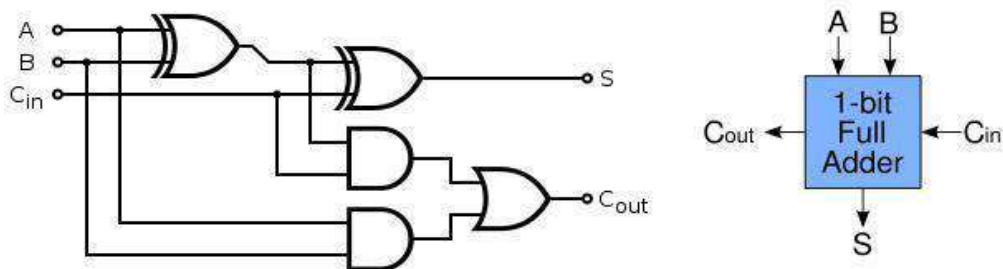


Figure 2. Full Adder (a) Logic Diagram and (b) Graphic Diagram

Subtractor

In electronics, a subtractor can be designed using the same approach as that of an adder. The binary subtraction process is summarized below. As with an adder, in the general case of calculations on multi-bit numbers, three bits are involved in performing the subtraction for each bit: the minuend (X_i), subtrahend (Y_i), and a borrow in from the previous (less significant) bit order position (B_i). The outputs are the difference bit (D_i) and borrow bit B_{i+1} .

Half subtractor

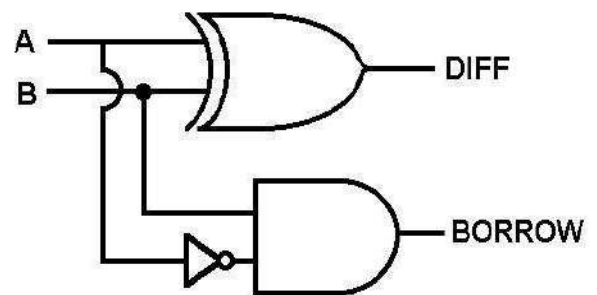
The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, X (minuend) and Y (subtrahend) and two outputs D (difference) and B (borrow). Such a circuit is called a half-subtractor because it enables a borrow out of the current arithmetic operation but no borrow in from a previous arithmetic operation.

The truth table for the half subtractor is given below.

X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$D = X'Y + XY' \quad \text{or} \quad D = X \oplus Y$$

$$B = X'Y$$



(a) Truth Table of half subtractor

(b) Logic Diagram of half subtractor

Figure 3. (a) Truth Table and (b) Logic Diagram of Half Subtractor

Full Subtractor

As in the case of the addition using logic gates, a *full subtractor* is made by combining two half-subtractors and an additional OR-gate. A full subtractor has the borrow in capability (denoted as BOR_{IN} in the diagram below) and so allows *cascading* which results in the possibility of **multi-bit subtraction**.

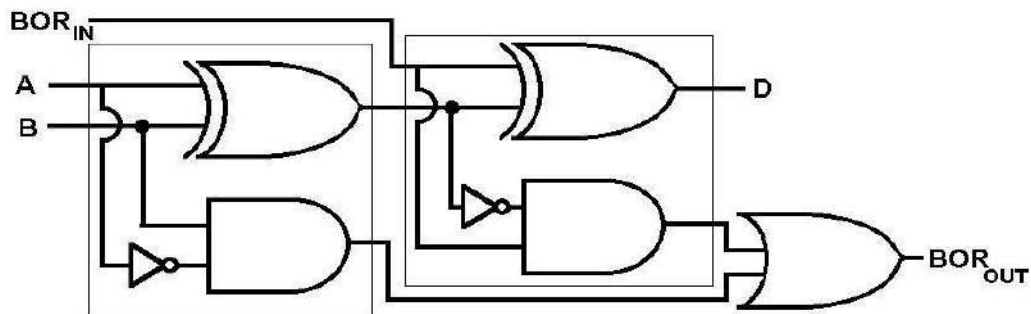
The final truth table for a full subtractor looks like:

A	B	BOR _{IN}	D	BOR _{OUT}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

(a) Truth Table of half subtractor

Find out the equations of the borrow and the difference

The circuit diagram for a full subtractor is given below.



(b) Logic Diagram of full subtractor

Figure 4. (a) Truth Table and (b) Logic Diagram of Full Subtractor

For a wide range of operations many circuit elements will be required. A neater solution will be to use subtraction via addition using *complementing* as was discussed in the binary arithmetic topic. In this case only adders are needed as shown bellow.

Lecture 6

Carry look ahead adder and Parity Generator

Carry Look Ahead Adder

The Carry Look Ahead removes the carry-ripple effect in the other types of adders described above. The architecture of a CLA is shown in Figure 5. Since the CLA generates a carry for each bit simultaneously, the delay is greatly reduced. The independent carry can be computed by expanding from equation 4:

$$C_{o,k} = G_k + P_k(G_{k-1} + P_{k-1}(\dots + P_1(G_0 + P_0C_i, 0))) \quad (10)$$

In practice, it is not possible to use the CLA to realize constant delay for the wider-bit adders since there will be a substantial loading capacitance, and hence larger delay and larger power consumption. The CLA has the fastest growing area requirements with respect to the bit size.

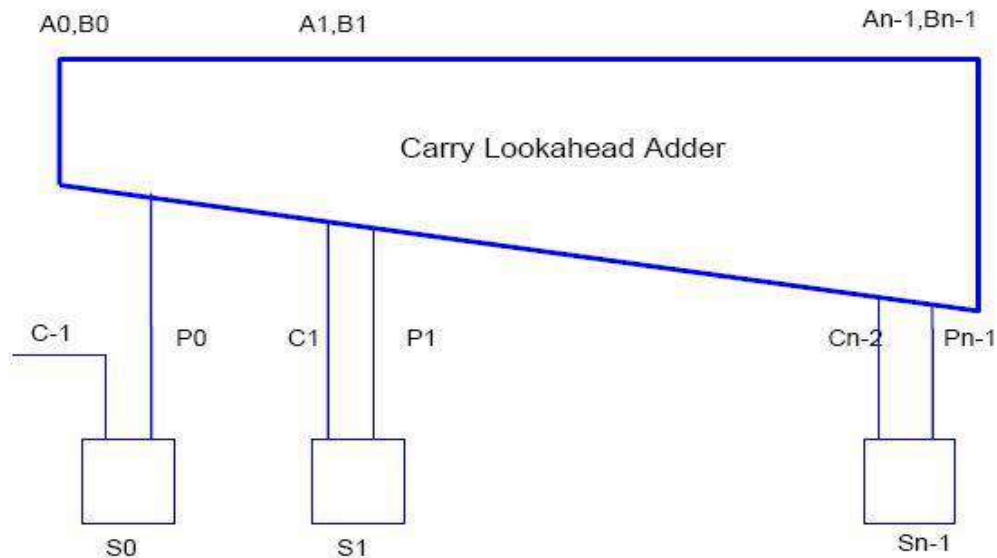


Figure 5: CLA architecture

In practice, the mixture of architectures listed above are often used in the design of wider bit adders to realize better optimization in terms of design metrics such as performance, power consumption, Power-delay Product (PDP), Energy-delay Product (EDP), area, and noise margin, etc.

One Bit Full Adder:

The main objective of this phase is to get a logically minimized expression to evaluate the two required output from the three inputs of the single bit full adder. First the main functionality of the single bit adder can be modeled by the following table (see table1)

Table 1 : One Bit Full Adder Truth Table

A	B	CIN	SUM	CO UT
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

Using simple logic design and minimization techniques we can derive the following formula for our single bit full adder.

$$\text{Sum} = (A \text{ XOR } B) \text{ XOR } \text{Cin}$$

$$\text{Cout} = A \cdot B + \text{Cin} \cdot (A \text{ XOR } B)$$

See the figure below for the gate implementation of this equation.

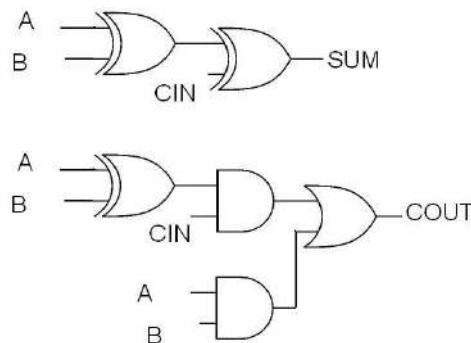


Figure 6: One bit Full Adder Logic Design

We can see clearly from **figure 1** that the critical path is the path from the input to the COUT. This gives us a clue for the path that we have to optimize during the circuit design phase. We have used (**Logic Works version 4.01**) to logically simulate the functionality of this adder. We applied the 8 test vectors shown above and ran the simulation for enough time. Finally we got the results witch matched the ones in **table1**.

Four Bit Full Adder:

Here is an important decision point which is affecting our design criteria. We need to choose the type of adder and the type of carry propagation we are going to use. There are two main categories of adders. Carry Look Ahead Adders and Carry Ripple Adders.

Ripple Carry Adder

In this type of adders, the output carry from one block is connected to the carry input pin of the next block. So the carry will propagate serially through full adder blocks (see **figure2**). The main advantage of this approach is its ease of implementation (modularity) but it cost more delay since the critical path length now is proportional to the size of the adder (number of bits) (see **figure2**) critical path is in red color.

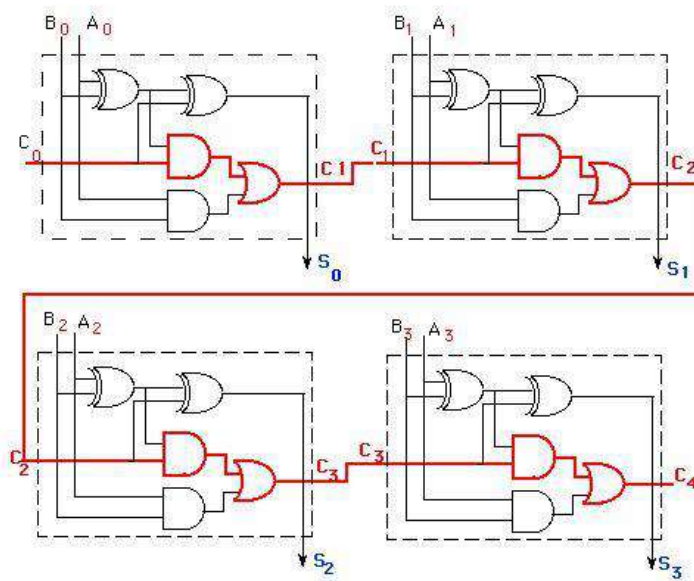


Figure 7: One bit Full Adder Logic Design

Carry Look Ahead adder

Carry look ahead adder depends on the idea of generating and propagating the carry. However the maximum delay for the carry now is 2 logic levels. See **figure 3** and **figure 4** for details.

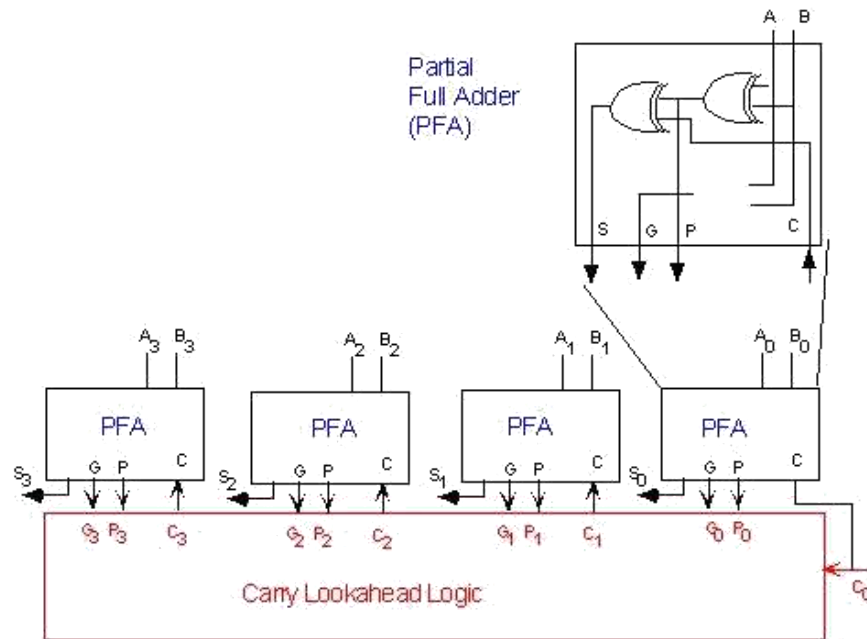


Figure 8. Carry Look Ahead Adder (Logical design)

Where G, P and carry values are as follows :

$$G_i = A_i \cdot B_i$$

$$P_i = (A_i \oplus B_i)$$

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

It is clear that the main advantage of Carry look a head adder is the delay reduction due to the Carry Look Ahead logic block. However it increases the area significantly as the number of bits increases .

Some other adder implementations also exists where different combinations of ripple and look ahead and other logic blocks are used.

Parity Generator

What is parity bit

The parity generating technique is one of the most widely used **error detection techniques** for the data transmission.

In digital systems, when binary data is transmitted and processed, data may be subjected to noise so that such noise can alter 0s (of data bits) to 1s and 1s to 0s.

Hence, **parity bit** is added to the word containing data in order to make number of 1s either even or odd. Thus it is used to detect errors, during the transmission of binary data. The message containing the data bits along with parity bit is transmitted from transmitter node to receiver node. At the receiving end, the number of 1s in the message **is counted and if it doesn't match with the transmitted one, then it means** there is an error in the data.

Parity Generator And Checker

A *parity generator* is a combinational logic circuit that generates the parity bit in the transmitter.

On the other hand, a circuit that checks the parity in the receiver is called parity checker. A combined circuit or devices of parity generators and parity checkers are commonly used in digital systems to detect the single bit errors in the transmitted data word.

The sum of the data bits and parity bits can be even or odd. In even parity, the added parity bit will make the total number of 1s an even amount whereas in odd parity the added parity bit will make the total number of 1s odd amount.

The basic principle involved in the implementation of parity circuits is that sum of odd number of 1s is always 1 and sum of even number of 1s is always zero. Such error detecting and correction can be implemented by using Ex-OR gates (since Ex-OR gate produce zero output when there are even number of inputs).

To produce two bits sum, one Ex-OR gate is sufficient whereas for adding three bits two Ex-OR gates are required as shown in below figure.

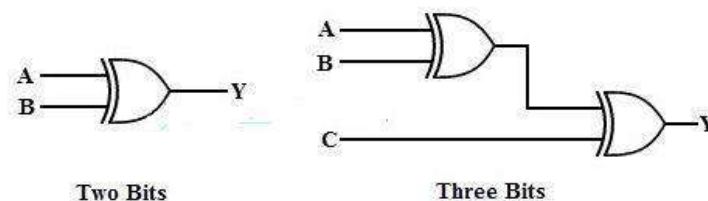


Figure 9: Ex-OR gates Parity Generator

It is combinational circuit that accepts an n-1 bit stream data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is termed as a parity bit. In **even parity bit scheme, the parity bit is '0' if there are even number of 1s** in the data stream and the **parity bit is '1' if there are odd number of 1s** in the data stream. In **odd parity bit scheme, the parity bit is '1' if there are even number of 1s in the data stream and the parity bit is '0' if there are odd number of 1s** in the data stream. Let us discuss both even and odd parity generators.

Even Parity Generator

Let us assume that a 3-bit message is to be transmitted with an even parity bit. Let the three inputs A, B and C are applied to the circuits and output bit is the parity bit P. The total number of 1s must be even, to generate the even parity bit P. The figure below shows the truth table of even parity generator in which 1 is placed as parity bit in order to make all 1s as even when the number of 1s in the truth table is odd.

Table 2. Even Parity Bit Generator

3-bit message			Even parity bit generator (P)
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The K-map simplification for 3-bit message even parity generator is

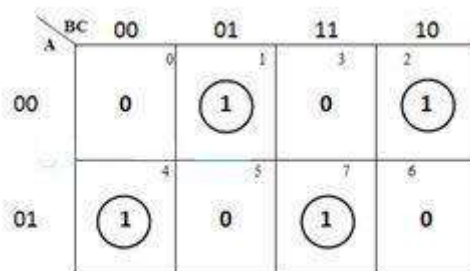


Figure 10. k- map of even parity bit generator

The logic diagram of even parity generator with two Ex – OR gates is shown below. The three bit message along with the parity generated by this circuit which is transmitted to the receiving end where

parity checker circuit checks whether any error is present or not. To generate the even parity bit for a 4-bit data, three Ex-OR gates are required to add the 4-bits and their sum will be the parity bit.

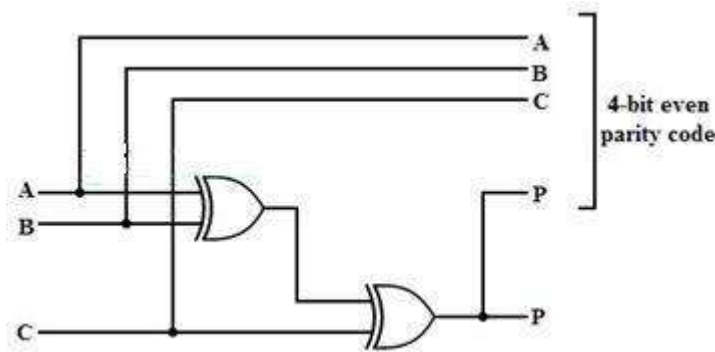


Figure 11. Circuit Diagram of even parity bit generator

Odd Parity Generator

Let us consider that the 3-bit data is to be transmitted with an odd parity bit. The three inputs are A, B and C and P is the output parity bit. The total number of bits must be odd in order to generate the odd parity bit. In the given truth table below, 1 is placed in the parity bit in order to make the total number of bits odd when the total number of 1s in the truth table is even.

Table 3. Odd Parity Bit Generator

3-bit message			Odd parity bit generator (P)
A	B	C	Y
0	0	0	1
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	0

The truth table of the odd parity generator can be simplified by using K-map as

	BC	00	01	11	10
A		0	1	3	2
00		1	0	1	0
01		0	1	0	1
		4	5	7	6

Figure 12. Odd Parity Bit Generator K-Map

The output parity bit expression for this generator circuit is obtained as

$$P = A \oplus B \text{ Ex-NOR } C$$

The above Boolean expression can be implemented by using one Ex-OR gate and one Ex-NOR gate in order to design a 3-bit odd parity generator. The logic circuit of this generator is shown in below figure , in which . two inputs are applied at one Ex-OR gate, and this Ex-OR output and third input is applied to the Ex-NOR gate , to produce the odd parity bit. It is also possible to design this circuit by using two Ex-OR gates and one NOT gate.

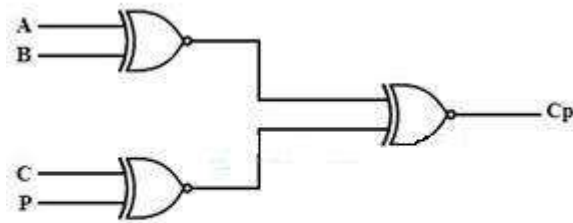


Figure 13. Circuit Diagram of odd parity bit generator

Parity Check

It is a logic circuit that checks for possible errors in the transmission.

This circuit can be an even parity checker or odd parity checker depending on the type of parity generated at the transmission end. When this circuit is used as even parity checker, the number of input bits must always be even. When a parity **error occurs, the 'sum even' output goes low and 'sum odd' output goes high**. If this logic circuit is used as an odd parity checker, the number of input bits should be odd, but if an **error occurs the 'sum odd' output goes low and 'sum even' output goes high**.

Even Parity Checker

Consider that three input message along with even parity bit is generated at the transmitting end. These 4 bits are applied as input to the parity checker circuit which checks the possibility of error on the data. Since the data is transmitted with even parity, four bits received at circuit must have an even number of 1s.

If any error occurs, the received message consists of odd number of 1s. The output of the parity checker is denoted by PEC (parity error check).

The below table shows the truth table for the even parity checker in which $PEC = 1$ if the error occurs, i.e., the four bits received have odd number of 1s and $PEC = 0$ if no error occurs, i.e., if the 4-bit message has even number of 1s.

Table 4. Even Parity Checker

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

The above truth table can be simplified using K-map as shown below.

	CP	00	01	11	10
AB		0 ⁰	1 ¹	3 ³	2 ²
00		0	1	0	1
01		4 ⁴	5 ⁵	7 ⁷	6 ⁶
		1	0	1	0
11		12 ¹²	13 ¹³	15 ¹⁵	14 ¹⁴
		0	1	0	1
10		8 ⁸	9 ⁹	11 ¹¹	10 ¹⁰
		1	0	1	0

Figure 14. K-map of even parity checker

$$\begin{aligned}
 \text{PEC} &= \bar{A} \bar{B} (\bar{C} D + C \bar{D}) + \bar{A} B (\bar{C} \bar{D} + C D) + A B (\bar{C} D + C \bar{D}) + A \bar{B} (\bar{C} \bar{D} + C D) \\
 &= \bar{A} \bar{B} (C \oplus D) + \bar{A} B (\overline{C \oplus D}) + A B (C \oplus D) + A \bar{B} (\overline{C \oplus D}) \\
 &= (\bar{A} \bar{B} + A B) (C \oplus D) + (\bar{A} B + A \bar{B}) (\overline{C \oplus D}) \\
 &= (A \oplus B) \oplus (C \oplus D)
 \end{aligned}$$

The above logic expression for the even parity checker can be implemented by using three Ex-OR gates as shown in figure. If the received message consists of five bits, then one more Ex-OR gate is required for the even parity checking.

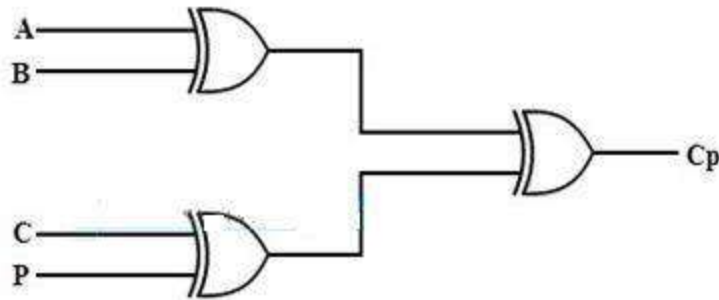


Figure 15. Logic Diagram of even parity checker

Odd Parity Checker

Consider that a three bit message along with odd parity bit is transmitted at the transmitting end. Odd parity checker circuit receives these 4 bits and checks whether any error are present in the data. If the

total number of 1s in the data is odd, then it indicates no error, whereas if the total number of 1s is even then it indicates the error since the data is transmitted with odd parity at transmitting end.

The below figure shows the truth table for odd parity generator where **PEC = 1** if the 4-bit message received consists of **even number of 1s** (hence the error occurred) and **PEC = 0** if the message contains **odd number of 1s** (that means no error).

Table 5. Odd Parity Checker

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

The expression for the PEC in the above truth table can be simplified by K-map as shown below.

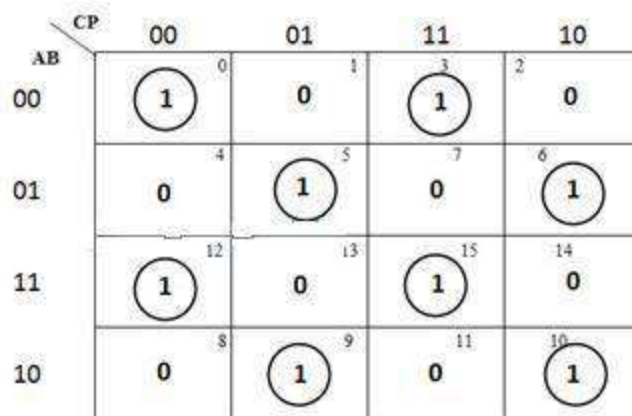


Figure 16. K-map of odd parity checker

After simplification, the final expression for the PEC is obtained as

$$PEC = (A \text{ Ex-NOR } B) \text{ Ex-NOR } (C \text{ Ex-NOR } D)$$

The expression for the odd parity checker can be designed by using three Ex-NOR gates as shown below.

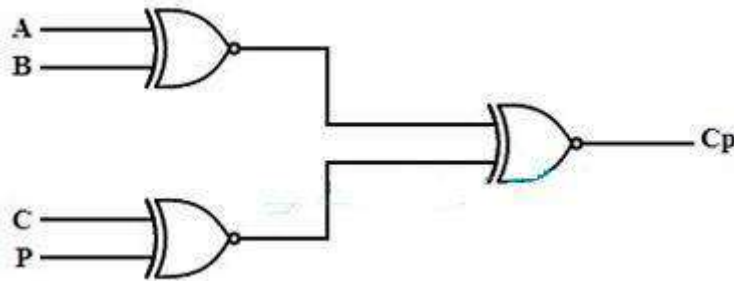


Figure 17. Logic Diagram of odd parity checker

Parity Generator/Checker ICs

There are different types of parity generator /checker ICs are available with different input configurations such as 5-bit, 4-bit, 9-bit, 12-bit, etc.

A most commonly used and standard type of parity generator/checker IC is 74180. It is a 9-bit parity generator or checker used to detect errors in high speed data transmission or data retrieval systems. The figure below shows the pin diagram of 74180 IC. This IC can be used to generate a 9-bit odd or even parity code or it can be used to check for odd or even parity in a 9-bit code (8 data bits and one parity bit).

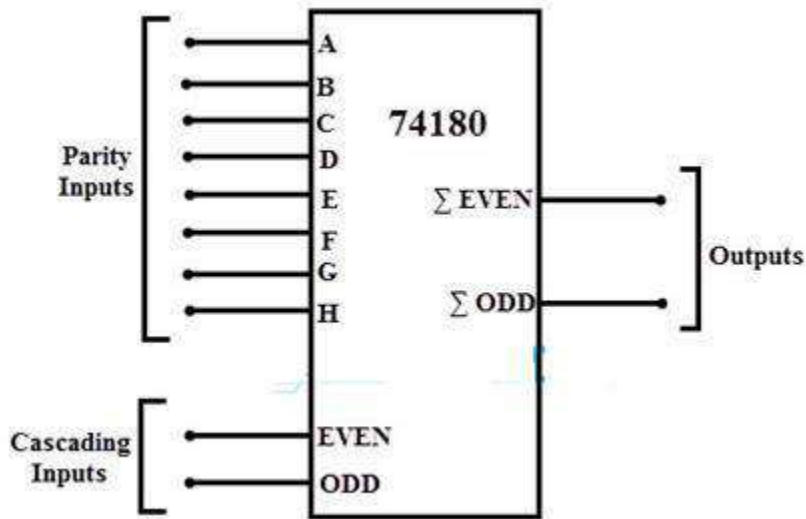


Figure 18. Graphic Diagram of Parity Generator

This IC consists of eight parity inputs from A through H and two cascading inputs. There are two outputs even sum and odd sum. In implementing generator or checker circuits, unused parity bits must be tied to logic zero and the cascading inputs must not be equal.

Lecture 7 & Lecture 8

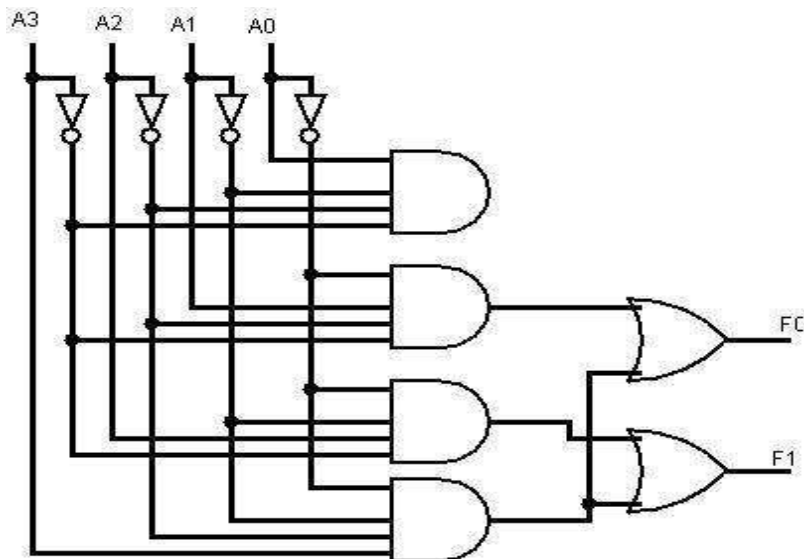
Encoder, Decoder, Multiplexer

Encoder

An encoder has 2^n input lines and n output lines. The output lines generate a binary code corresponding to the input value. For example a single bit 4 to 2 encoder takes in 4 bits and outputs 2 bits. It is assumed that there are only 4 types of input signals these are : 0001, 0010, 0100, 1000.

I ₃	I ₂	I ₁	I ₀	F ₁	F ₀
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth Table of 4 to 2 encoder



(b) Circuit Diagram of encoder

Figure 19. (a) Truth Tablee and (b) Circuit Diagram of Encoder

The encoder has the limitation that only one input can be active at any given time. If two inputs are simultaneously active, the output produces an undefined combination. To prevent this we make use of

the priority encoder.

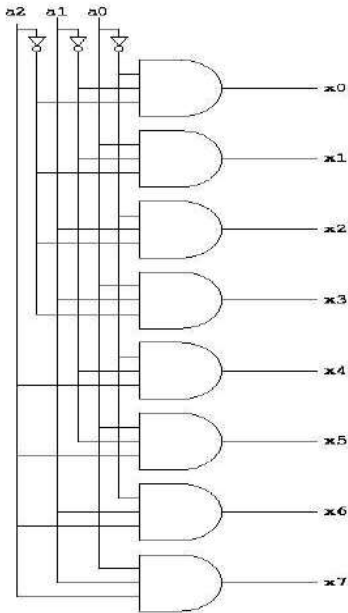
Decoder

In both the multiplexer and the demultiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of n digits to 2^n outputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0.

It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*. It has the following truth table (for $n = 3$):

a2	a1	a0	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

(a) Truth Table of decoder



(b) circuit diagram for the decoder

Figure 20. (a) Truth Table and (b) Circuit Diagram of Decoder

Multiplexer

A multiplexer is a combinatorial circuit that is given a certain number (usually a power of two) *data inputs*, let us say 2^n , and n *address inputs* used as a binary number to select one of the data inputs. The multiplexer has a single output, which has the same value as the selected data input.

In other words, the multiplexer works like the input selector of a home music system. Only one input is selected at a time, and the selected input is transmitted to the single output. While on the music system, the selection of the input is made manually, the multiplexer chooses its input based on a binary number, the address input.

The truth table for a multiplexer is huge for all but the smallest values of n . We therefore use an abbreviated version of the truth table in which some inputs are replaced by '-' to indicate that the input value does not matter.

Here is such an abbreviated truth table for $n = 3$. The full truth table would have $2^{(3 + 23)} = 2048$ rows.

<u>SELECT</u>			<u>INPUT</u>										
a2	a1	a0	d7	d6	d5	d4	d3	d2	d1	d0		x	
-	-	-	-	-	-	-	-	-	-	-	-	----	
0	0	0	-	-	-	-	-	-	-	-	0	0	
0	0	0	-	-	-	-	-	-	-	-	1	1	
0	0	1	-	-	-	-	-	-	-	-	0	- 0	
0	0	1	-	-	-	-	-	-	-	-	1	- 1	
0	1	0	-	-	-	-	-	-	-	-	0	- - 0	
0	1	0	-	-	-	-	-	-	-	-	1	- - 1	
0	1	1	-	-	-	-	-	-	-	-	0	- - - 0	
0	1	1	-	-	-	-	-	-	-	-	1	- - - 1	
1	0	0	-	-	-	-	-	-	-	-	0	- - - - 0	
1	0	0	-	-	-	-	-	-	-	-	1	- - - - 1	
1	0	1	-	-	0	-	-	-	-	-	-	- 0	
1	0	1	-	-	1	-	-	-	-	-	-	- 1	
1	1	0	-	0	-	-	-	-	-	-	-	- 0	
1	1	0	-	1	-	-	-	-	-	-	-	- 1	
1	1	1	0	-	-	-	-	-	-	-	-	- 0	
1	1	1	1	-	-	-	-	-	-	-	-	- 1	

(a) Truth Table of multiplexer

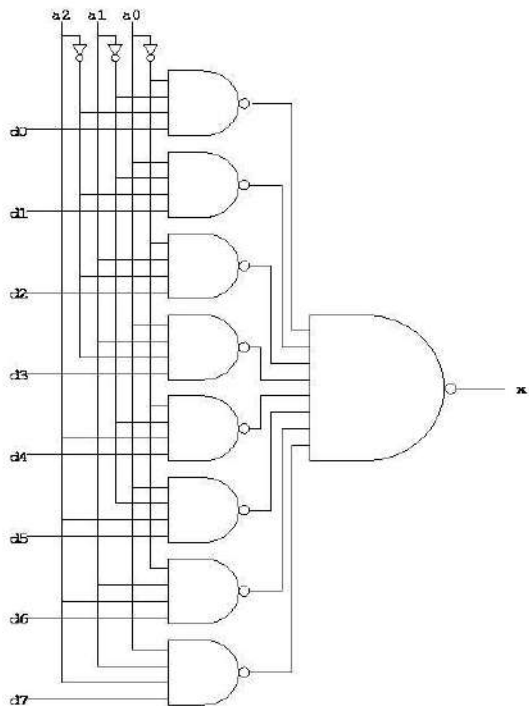
We can abbreviate this table even more by using a letter to indicate the value of the selected input, like

this:

a2	a1	a0	d7	d6	d5	d4	d3	d2	d1	d0	x
-	-	-	-	-	-	-	-	-	-	-	-
0	0	0	-	-	-	-	-	c	-	-	c
0	0	1	-	-	-	-	-	c	-	-	c
0	1	0	-	-	-	-	-	c	-	-	c
0	1	1	-	-	-	-	c	-	-	-	c
1	0	0	-	-	-	c	-	-	-	-	c
1	0	1	-	-	c	-	-	-	-	-	c
1	1	0	-	c	-	-	-	-	-	-	c
1	1	1	c	-	-	-	-	-	-	-	c

(b) Truth Table of multiplexer

The same way we can simplify the truth table for the multiplexer, we can also simplify the corresponding circuit. Indeed, our simple design method would yield a very large circuit. The simplified circuit looks like this:



(c) Circuit Diagram of multiplexer

Figure 21. Truth Table and Circuit Diagram of Multiplexer

Lecture 9

De-Multiplexer ,Comparator[[1L]

Magnitude Comparator

A magnitude comparator is a combinational circuit that compares two numbers A & B to determine whether:

$$A > B, \text{ or } A = B, \text{ or } A < B$$

Inputs :

First n -bit number A

Second n -bit number B

Outputs :

3 output signals (GT, EQ, LT), where:

$$3. \text{ GT} = 1 \text{ IFF } A > B$$

$$4. \text{ EQ} = 1 \text{ IFF } A = B$$

$$\text{LT} = 1 \text{ IFF } A < B$$

It must be noted Exactly One of these 3 outputs equals 1, while the other 2 outputs are 0's.

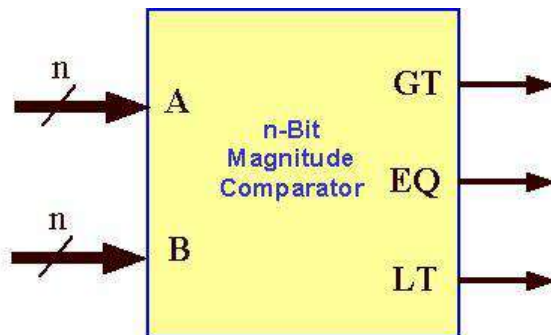


Figure 22. Graphic Diagram of n-bit Magnitude Comparator

4-bit magnitude \Rightarrow comparator \Rightarrow

Inputs: 8-bits (A 4-bits , B 4-bits)

A and B are two 4-bit numbers

Let $A = A_3A_2A_1A_0$, and

Let $B = B_3B_2B_1B_0$

Inputs have 28 (256) possible combinations

Not easy to design using conventional technique.

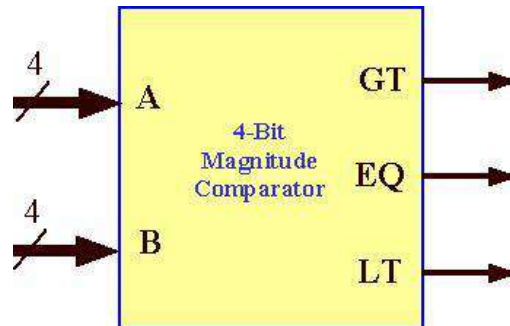


Figure 23. Graphic Diagram of 4-bit Magnitude Comparator

The circuit possesses certain amount of regularity \Rightarrow *can be designed algorithmically*. Design of the EQ output ($A = B$) in 4-bit magnitude comparator

Define $X_i = (A_i B_i) + (A_i$

$\neg B_i$

$\neg)$

Thus $X_i = 1$ IFF $A_i = B_i \quad i=0, 1, 2$ and 3

$X_i = 0$ IFF $A_i \neq B_i \quad \forall$

Condition for $A = B$

$EQ = 1$ (i.e., $A = B$) IFF

7. $A_3 = B_3 \rightarrow (X_3 = 1)$, and

8. $A_2 = B_2 \rightarrow (X_2 = 1)$, and

9. $A_1 = B_1 \rightarrow (X_1 = 1)$, and

10. $A_0 = B_0 \rightarrow (X_0 = 1)$.

Thus, $EQ = 1$ IFF $X_3 X_2 X_1 X_0 = 1$. In other words, $EQ = X_3 X_2 X_1 X_0$

Design of the GT output ($A > B$) 4-bit magnitude comparator

If $A_3 > B_3$, then $A > B$ ($GT=1$) irrespective of the relative values of the other bits of A & B. Consider, for example, $A = 1000$ and $B = 0111$ where $A > B$. This can be stated as $GT=1$ if $A_3 B_3' = 1$

If $A_3 = B_3$ ($X_3 = 1$), we compare the next significant pair of bits (A_2 & B_2).

If $A_2 > B_2$ then $A > B$ ($GT=1$) irrespective of the relative values of the other bits of A & B. Consider, for example, $A = 0100$ and $B = 0011$ where $A > B$. This can be stated as $GT=1$ if $X_3 A_2 B_2' = 1$

If $A_3 = B_3$ ($X_3 = 1$) and $A_2 = B_2$ ($X_2 = 1$), we compare the next significant pair of bits (A_1 & B_1).

If $A_1 > B_1$ then $A > B$ ($GT=1$) irrespective of the relative values of the remaining bits A_0 & B_0 . Consider, for example, $A = 0010$ and $B = 0001$ where $A > B$. This can be stated as $GT=1$ if $X_3 X_2 A_1 B_1' = 1$

If $A_3 = B_3$ ($X_3 = 1$) and $A_2 = B_2$ ($X_2 = 1$) and $A_1 = B_1$ ($X_1 = 1$), we compare the next pair of bits (A_0 & B_0).

If $A_0 > B_0$ then $A > B$ ($GT=1$). This can be stated as $GT=1$ if $X_3 X_2 X_1 A_0 B_0' = 1$

To summarize, $GT = 1$ ($A > B$) IFF:

$$= A_3 B_3' = 1, \text{ or}$$

$$= X_3 A_2 B_2' = 1, \text{ or}$$

$$= X_3 X_2 A_1 B_1' = 1, \text{ or}$$

$$= X_3 X_2 X_1 A_0 B_0' = 1$$

In other words, $GT = A_3 B_3' + X_3 A_2 B_2' + X_3 X_2 A_1 B_1' + X_3 X_2 X_1 A_0 B_0'$

Design of the LT output ($A < B$) 4-bit magnitude comparator

In the same manner as above, we can derive the expression of the LT ($A < B$) output

$$LT = B_3 A_3' + X_3 B_2 A_2' + X_3 X_2 B_1 A_1' + X_3 X_2 X_1 B_0 A_0'$$

The gate implementation of the three output variables (EQ, GT & LT) is shown in the figure below.

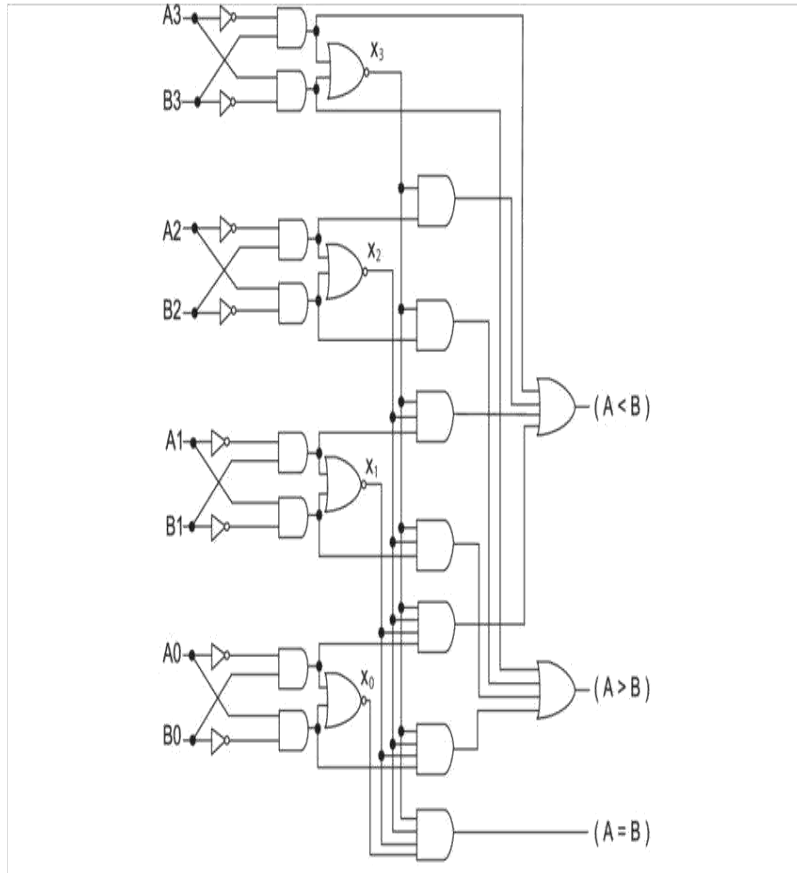


Figure 24. Circuit Diagram of Comparator

Modification to the Design

The hardware in the comparator can be reduced by implementing only two outputs, and the third output can be obtained using these two outputs. For example, if we have the LT and GT outputs, then the EQ output can be obtained by using only a NOR gate, as shown in the figure below.

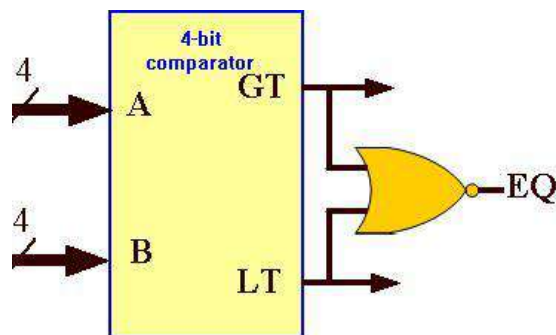


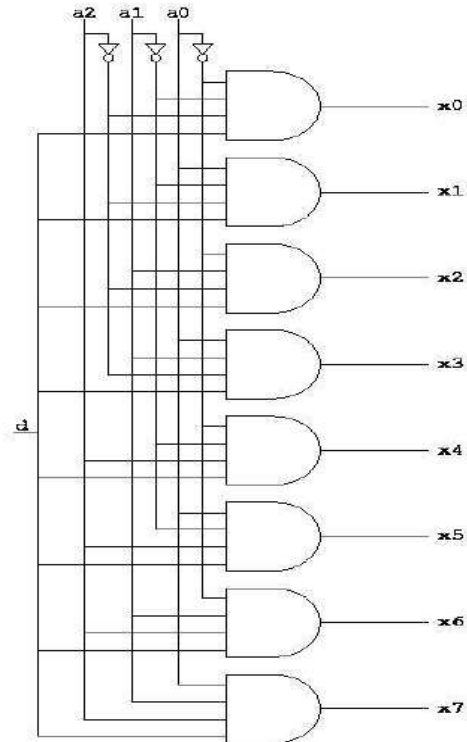
Figure 25. Graphic Diagram of modified 4-bit Magnitude Comparator

Thus, when both the GT and LT outputs are zeros, then the 3rd one (i.e. EQ) is a '1'.

Demultiplexer

The de-multiplexer is the inverse of the multiplexer, in that it takes a single data input and n address inputs. It has 2^n outputs. The address input determine which data output is going to have the same value as the data input. The other data outputs will have the value 0. Here is an abbreviated truth table for the de-multiplexer. We could have given the full table since it has only 16 rows, but we will use the same convention as for the multiplexer where we abbreviated the values of the data inputs.

a2	a1	a0	d	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	c	0	0	0	0	0	0	0	0
0	0	0	0	c	0	0	0	0	1	0	0
c	0	0	0	0	0	0	0	0	0	0	0
c	0	0	1	0	c	0	0	0	0	0	0
0	0	0	0	c	0	0	0	0	0	0	0
1	1	c	0	0	0	0	0	0	0	0	0
c	0	0	0	1	0	0	0	c	0	0	0
0	0	0	0	c	0	0	0	0	0	0	0
0	1	0	1	c	0	0	0	0	0	0	0
c	0	0	0	0	0	1	1	1	1	c	0
0	c	0	0	c	0	0	0	0	0	0	0
0	0	0	1	1	1	1	c	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0	0



(a) Truth table for the demultiplexer

(b) circuit diagram for the demultiplexer

Figure 26. (a) Truth Table and (b) Circuit Diagram of Demultiplexer

Lecture 10

Basic Concepts of A/D and D/A converters[1L]

Analog to Digital Converters (A/D)

An electronic integrated circuit which transforms a signal from analog (continuous) to digital (discrete) form. Analog signals are directly measurable quantities. Digital signals only have two states. For digital computer, we refer to binary states, 0 and 1.

Need For A/D Converter

Microprocessors can only perform complex processing on digitized signals. When signals are in digital form they are less susceptible to the deleterious effects of additive noise. ADC Provides a link between the analog world of transducers and the digital world of signal processing and data handling.

Application Of A/D Converter

ADC are used virtually everywhere where analog signal has to be processed, stored, or transported in digital form. Some examples of ADC usage are digital voltmeters, cell phone, thermocouples, an digital oscilloscope. Microcontrollers commonly use 8, 10, 12, or 16 bit ADCs, our micro controller uses an 8 or 10 bit ADC.

The ADC Process

ADC is a two step process: **Sampling and Holding (S/H)** and **Quantizing and Encoding (Q/E)**.

Sampling and Holding (S/H)

Holding signal benefits the accuracy of the A/D conversion. Minimum sampling rate should be at least twice the highest data frequency of the analog signal.

Quantizing and Encoding (Q/E)

Resolution:

The smallest change in analog signal that will result in a change in the digital output.

V = Reference voltage range

N = Number of bits in digital output.

2^N = Number of states.

ΔV = Resolution

The resolution represents the quantization error inherent in the conversion of the signal to digital form.

Quantizing: Partitioning the reference signal range into a number of discrete quanta, then matching the input signal to the correct quantum.

Encoding: Assigning a unique digital code to each quantum, then allocating the digital code to the input signal.

Types of A/D Converters

- 1) Dual Slope A/D Converter
- 2) Successive Approximation A/D Converter
- 3) Flash A/D Converter
- 4) Delta-Sigma A/D Converter

Other, Voltage-to-frequency, staircase ramp or single slope, charge balancing or redistribution, switched capacitor, tracking, and resolver.

ADC Types Comparison

Table 1. ADC Type Comparison Table

Type	Speed (relative)	Cost (relative)
Dual slope	Slow	Med
Flash	Very fast	High
Successive Approx	Medium – Fast	Low
Sigma-Delta	Slow	Low

Digital to Analog Converters (D/A)

The process of converting digital signal into equivalent analog signal is called D/A conversion. The electronics circuit, which does this process, is called D/A converter. **The circuit has 'n' number of digital data inputs with only one output.** Basically, there are two types of D/A converter circuits: Weighted resistors D/A converter circuit and Binary ladder or R–2R ladder D/A converter circuit.

Weighted resistors D/A converter – here an opamp is used as summing amplifier. There are four resistors R, 2R, 4R and 8R at the input terminals of the opamp with R as feedback resistor. The network of resistors at the input terminal of opamp is called as variable resistor network. The four inputs of the circuit are D, C, B & A. Input D is at MSB and A is at LSB. Here we shall connect 8V DC voltage as logic–1 level. So we shall assume that 0 = 0V and 1 = 8V. Now the working of the circuit is as follows.

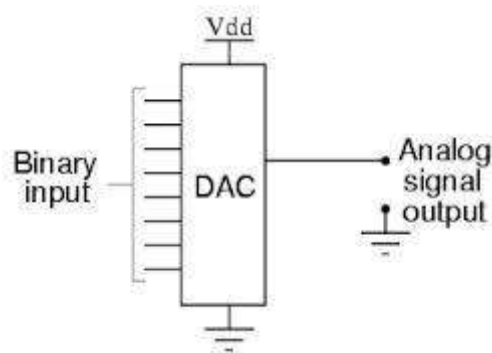


Figure 1. Graphic Diagram of D/A Converter

Application Of D/A Converter

Audio : DACs are commonly used in music players to convert digital data streams into analog audio signals.

Video : They are also used in televisions and mobile phones to convert digital video data into analog video signals which connect to the screen drivers to display monochrome or color images.

These two applications use DACs at opposite ends of the speed/resolution trade-off. The audio DAC is a low speed high resolution type while the video DAC is a high speed low to medium resolution type.

Discrete DACs would typically be extremely high speed low resolution power hungry types, as used in military radar systems. Very high speed test equipment, especially sampling oscilloscopes, may also use discrete DACs.

Types Of D/A Converter

The most common types of electronic DACs are:

- 1) The **pulse-width modulator**, the simplest DAC type.
- 2) The **delta-sigma DAC**.
- 3) The **R-2R ladder DAC**.
- 4) The **Successive-Approximation or Cyclic DAC**.
- 5) The **thermometer-coded DAC**.

Lecture 11,12 &13

Basic Flip-flop- SR, JK, D, T and JK Master-slave Flip Flops

Sequential circuit

Introduction

In order to build sophisticated digital logic circuits, including computers, we need more a powerful model. We need circuits whose output depends upon both the input of the circuit and its previous state. In other words, we need circuits that have *memory*.

For a device to serve as a memory, it must have three characteristics:

5. the device must have two stable states
6. there must be a way to read the state of the device
7. there must be a way to set the state at least once.

It is possible to produce circuits with memory using the digital logic gates we've already seen. To do that, we need to introduce the concept of *feedback*. So far, the logical flow in the circuits we've studied has been from input to output. Such a circuit is called *acyclic*. Now we will introduce a circuit in which the output is fed back to the input, giving the circuit memory. (There are other memory technologies that store electric charges or magnetic fields; these do not depend on feedback.)

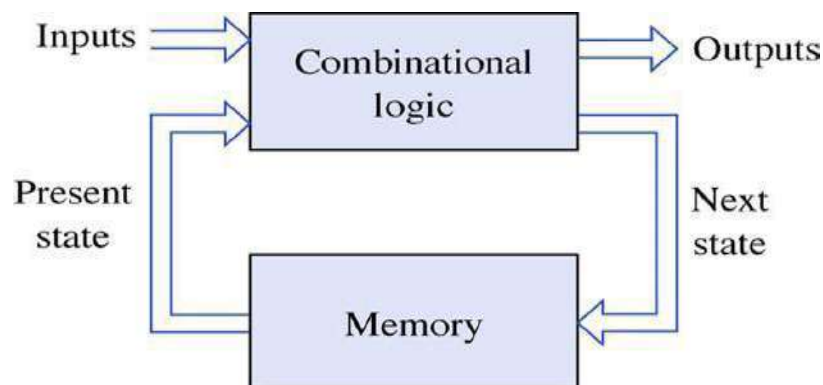


Figure 1

Latches and flip-flops

In the same way that gates are the building blocks of combinatorial circuits, *latches* and

flip-flops are the building blocks of sequential circuits.

While gates had to be built directly from transistors, latches can be built from gates, and flip-flops can be built from latches. This fact will make it somewhat easier to understand latches and flip-flops.

Both latches and flip-flops are circuit elements whose output depends not only on the current inputs, but also on previous inputs and outputs. The difference between a latch and a flip-flop is that a latch does not have a *clock signal*, whereas a flip-flop always does.

Latches

How can we make a circuit out of gates that is not combinatorial? The answer is *feed-back*, which means that we create *loops* in the circuit diagrams so that output values depend, indirectly, on themselves. If such feed-back is *positive* then the circuit tends to have stable states, and if it is *negative* the circuit will tend to oscillate.

In order for a logical circuit to "remember" and retain its logical state even after the controlling input signal(s) have been removed, it is necessary for the circuit to include some form of feedback. We might start with a pair of inverters, each having its input connected to the other's output. The two outputs will always have opposite logic level.

The problem with this is that we don't have any additional inputs that we can use to change the logic states if we want. We can solve this problem by replacing the inverters with NAND or NOR gates, and using the extra input lines to control the circuit.

The circuit shown below is a basic NAND latch. The inputs are generally designated "S" and "R" for "Set" and "Reset" respectively. Because the NAND inputs must normally be logic 1 to avoid affecting the latching action, the inputs are considered to be inverted in this circuit.

The outputs of any single-bit latch or memory are traditionally designated Q and Q'. In a commercial latch circuit, either or both of these may be available for use by other circuits. In any case, the circuit itself is:

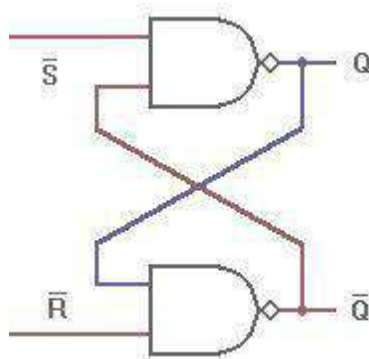


Figure 2

For the NAND latch circuit, both inputs should normally be at a logic 1 level. Changing an input to a logic 0 level will force that output to a logic 1. The same logic 1 will also be

applied to the second input of the other NAND gate, allowing that output to fall to a logic 0 level. This in turn feeds back to the second input of the original gate, forcing its output to remain at logic 1.

Applying another logic 0 input to the same gate will have no further effect on this circuit. However, applying a logic 0 to the *other* gate will cause the same reaction in the other direction, thus changing the state of the latch circuit the other way.

Note that it is forbidden to have both inputs at a logic 0 level at the same time. That state will force both outputs to a logic 1, overriding the feedback latching action. In this condition, whichever input goes to logic 1 first will lose control, while the other input (still at logic 0) controls the resulting state of the latch. If both inputs go to logic 1 simultaneously, the result is a "race" condition, and the final state of the latch cannot be determined ahead of time. The same functions can also be performed using NOR gates. A few adjustments must be made to allow for the difference in the logic function, but the logic involved is quite similar.

The circuit shown below is a basic NOR latch. The inputs are generally designated "S" and "R" for "Set" and "Reset" respectively. Because the NOR inputs must normally be logic 0 to avoid overriding the latching action, the inputs are not inverted in this circuit. The NOR-based latch circuit is:

For the NOR latch circuit, both inputs should normally be at a logic 0 level. Changing an input to a logic 1 level will force that output to a logic 0. The same logic 0 will also be applied to the second input of the other NOR gate, allowing that output to rise to a logic 1 level. This in turn feeds back to the second input of the original gate, forcing its output to remain at logic 0 even after the external input is removed.

Applying another logic 1 input to the same gate will have no further effect on this circuit. However, applying a logic 1 to the *other* gate will cause the same reaction in the other direction, thus changing the state of the latch circuit the other way.

Note that it is forbidden to have both inputs at a logic 1 level at the same time. That state will force both outputs to a logic 0, overriding the feedback latching action. In this condition, whichever input goes to logic 0 first will lose control, while the other input (still at logic 1) controls the resulting state of the latch. If both inputs go to logic 0 simultaneously, the result is a "race" condition, and the final state of the latch cannot be determined ahead of time.

One problem with the basic RS NOR latch is that the input signals actively drive their respective outputs to a logic 0, rather than to a logic 1. Thus, the S input signal is applied to the gate that produces the Q' output, while the R input signal is applied to the gate that produces the Q output. The circuit works fine, but this reversal of inputs can be confusing when you first try to deal with NOR-based circuits.

Flip-flops

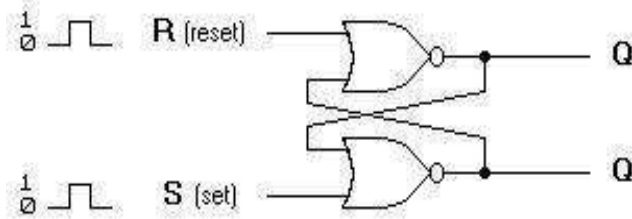
Latches are asynchronous, which means that the output changes very soon after the input changes. Most computers today, on the other hand, are synchronous, which means that the outputs of all the sequential circuits change simultaneously to the rhythm of a global clock signal.

A flip-flop is a synchronous version of the latch. A flip-flop circuit can be constructed from two NAND gates or two NOR gates. These flip-flops are shown in Figure 2 and Figure 3. Each flip-flop has two outputs, Q and Q', and two inputs, set and reset. This type of flip-flop is

referred to as an SR flip-flop or SR latch. The flip-flop in Figure 2 has two useful states. When $Q=1$ and $Q'=0$, it is in the set state

(or 1-state). When $Q=0$ and $Q'=1$, it is in the clear state (or 0 -state). The outputs Q and Q' are complements of each other and are referred to as the normal and complement outputs, respectively. The binary state of the flip-flop is taken to be the value of the normal output.

When a 1 is applied to both the set and reset inputs of the flip-flop in Figure 2, both Q and Q' outputs go to 0. This condition violates the fact that both outputs are complements of each other. In normal operation this condition must be avoided by making sure that 1's are not applied to both inputs simultaneously.

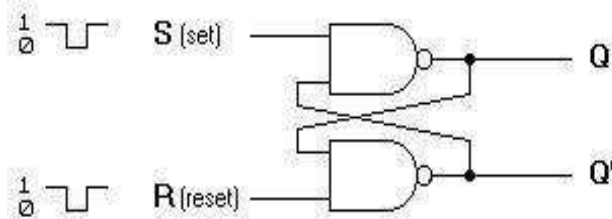


(a) Logic diagram

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

(b) Truth table

Figure 3. Basic flip-flop circuit with NAND gates



(a) Logic diagram

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

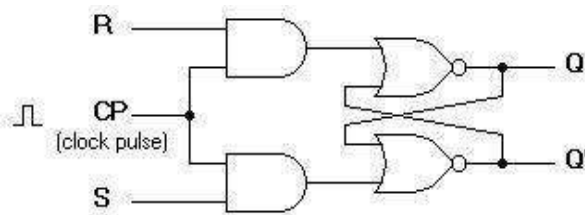
(b) Truth table

Figure 4. Basic flip-flop circuit with NAND gates

The NAND basic flip-flop circuit in Figure 3(a) operates with inputs normally at 1 unless the state of the flip-flop has to be changed. A 0 applied momentarily to the set input causes Q to go to 1 and Q' to go to 0, putting the flip-flop in the set state. When both inputs go to 0, both outputs go to 1. This condition should be avoided in normal operation.

Clocked SR Flip-Flop

The clocked SR flip-flop shown in Figure 4 consists of a basic NOR flip-flop and two AND gates. The outputs of the two AND gates remain at 0 as long as the clock pulse (or CP) is 0, regardless of the S and R input values. When the clock pulse goes to 1, information from the S and R inputs passes through to the basic flip-flop. With both S=1 and R=1, the occurrence of a clock pulse causes both outputs to momentarily go to 0. When the pulse is removed, the state of the flip-flop is indeterminate, i.e., either state may result, depending on whether the set or reset input of the flip-flop remains a 1 longer than the transition to 0 at the end of the pulse.



(a) Logic diagram

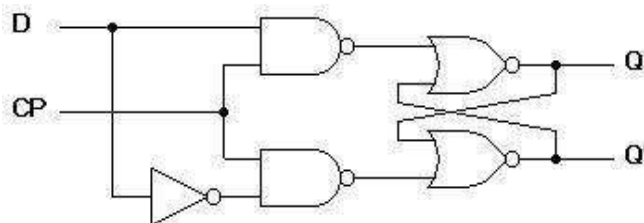
Q	S	R	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	indeterminate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	indeterminate

(b) Truth Table

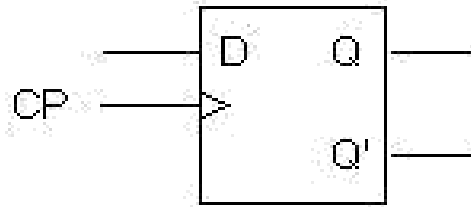
Figure 5. Clocked S R Flip Flop

D Flip-Flop

The D flip-flop shown in Figure 5 is a modification of the clocked SR flip-flop. The D input goes directly into the S input and the complement of the D input goes to the R input. The D input is sampled during the occurrence of a clock pulse. If it is 1, the flip-flop is switched to the set state (unless it was already set). If it is 0, the flip-flop switches to the clear state.



(a) Logic diagram with NAND gates



(b) Graphical symbol

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

(c) Transition table

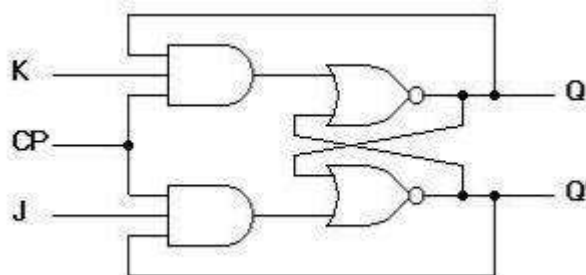
Figure 6. Clocked D flip-flop

JK Flip-Flop

A JK flip-flop is a refinement of the SR flip-flop in that the indeterminate state of the SR type is defined in the JK type. Inputs J and K behave like inputs S and R to set and clear the flip-flop (note that in a JK flip-flop, the letter J is for set and the letter K is for clear). When logic 1 inputs are applied to both J and K simultaneously, the flip-flop switches to its complement state, i.e., if $Q=1$, it switches to $Q=0$ and vice versa.

A clocked JK flip-flop is shown in figure below. Output Q is ANDed with K and CP inputs so that the flip-flop is cleared during a clock pulse only if Q was previously 1. Similarly, output **Q' is ANDed with J and CP inputs so that the flip-flop is set with a clock pulse only if Q' was previously 1.**

Note that because of the feedback connection in the JK flip-flop, a CP signal which remains a 1 (while $J=K=1$) after the outputs have been complemented once will cause repeated and continuous transitions of the outputs. To avoid this, the clock pulses must have a time duration less than the propagation delay through the flip-flop. The restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction. The same reasoning also applies to the T flip-flop presented next.



(a) Logic diagram

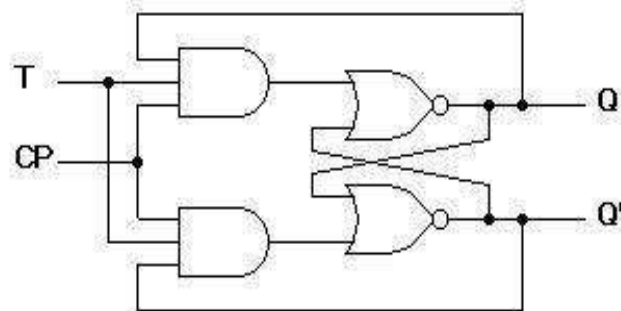
Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(b) Transition Table

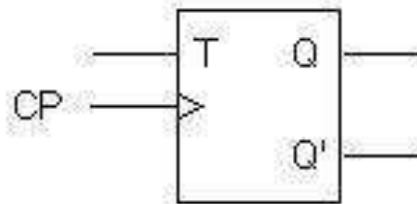
Figure 7. Clocked JK flip-flop

T Flip-Flop

The T flip-flop is a single input version of the JK flip-flop. As shown in figure below, the T flip-flop is obtained from the JK type if both inputs are tied together. The output of the T flip-flop "toggles" with each clock pulse.



(a) Logic diagram



(b) Graphical symbol

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

(c) Transition table

Figure 8. T Flip Flop

Triggering of Flip-flops

The state of a flip-flop is changed by a momentary change in the input signal. This change is called a trigger and the transition it causes is said to trigger the flip-flop. The basic circuits of Figure 2 and Figure 3 require an input trigger defined by a change in signal level. This level must be returned to its initial level before a second trigger is applied. Clocked flip-flops are triggered by pulses.

The feedback path between the combinational circuit and memory elements in Figure 1 can produce instability if the outputs of the memory elements (flip-flops) are changing while the outputs of the combinational circuit that go to the flip-flop inputs are being sampled by the clock pulse. A way to solve the feedback timing problem is to make the flip-flop sensitive to the pulse transition rather than the pulse duration.

The clock pulse goes through two signal transitions: from 0 to 1 and the return from 1 to 0. As shown in Figure, the positive transition is defined as the positive edge and the negative transition as the negative edge.

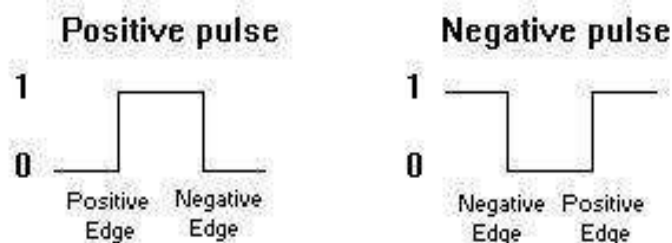


Figure 9 . Definition of clock pulse transition

The clocked flip-flops already introduced are triggered during the positive edge of the pulse, and the state transition starts as soon as the pulse reaches the logic-1 level. If the other inputs change while the clock is still 1, a new output state may occur. If the flip-flop is made to respond to the positive (or negative) edge transition only, instead of the entire pulse duration, then the multiple-transition problem can be eliminated.

Master-Slave Flip-Flop

A master-slave flip-flop is constructed from two separate flip-flops. One circuit serves as a master and the other as a slave. The logic diagram of an SR flip-flop is shown in Figure 9. The master flip-flop is enabled on the positive edge of the clock pulse CP and the slave flip-flop is disabled by the inverter. The information at the external R and S inputs is transmitted to the master flip-flop. When the pulse returns to 0, the master flip-flop is disabled and the slave flip-flop is enabled. The slave flip-flop then goes to the same state as the master flip-flop.

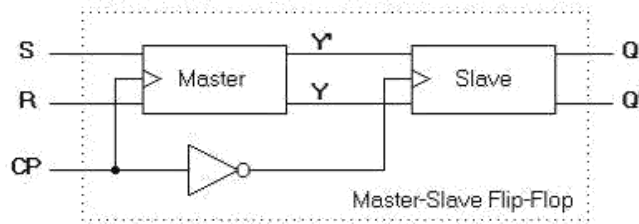
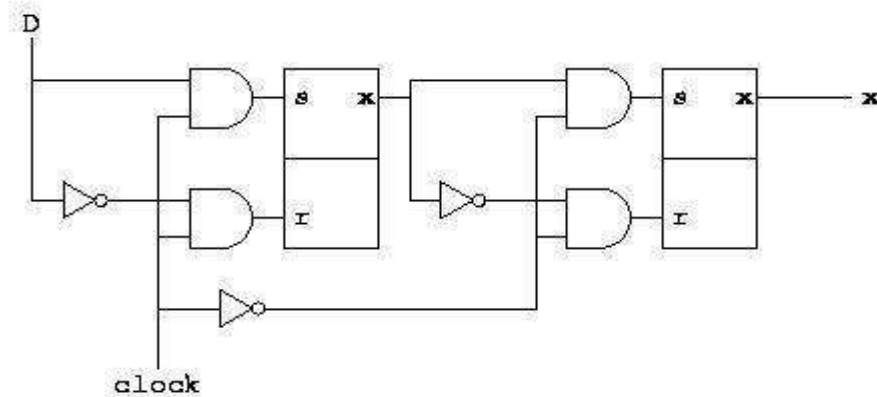


Figure 10. Logic diagram of a master-slave flip-flop



Master slave RS flip flop

The timing relationship is shown in Figure 11 and is assumed that the flip-flop is in the clear state prior to the occurrence of the clock pulse. The output state of the master-slave flip-flop occurs on the negative transition of the clock pulse. Some master-slave flip-flops change output state on the positive transition of the clock pulse by having an additional inverter between the CP terminal and the input of the master.

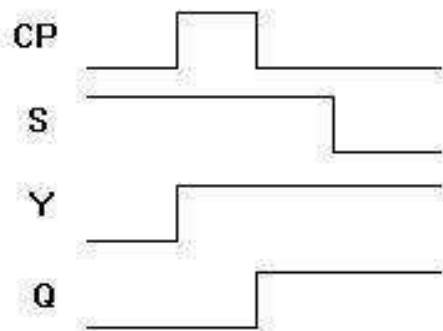


Figure 11. Timing relationship in a master slave flip-flop

Edge Triggered Flip-Flop

Another type of flip-flop that synchronizes the state changes during a clock pulse transition is the edge-triggered flip-flop. When the clock pulse input exceeds a specific threshold level, the inputs are locked out and the flip-flop is not affected by further changes in the inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the positive edge of the clock pulse (positive-edge-triggered), and others on the negative edge of the pulse (negative-edge-triggered). The logic diagram of a D-type positive-edge-triggered flip-flop is shown in Figure 12.

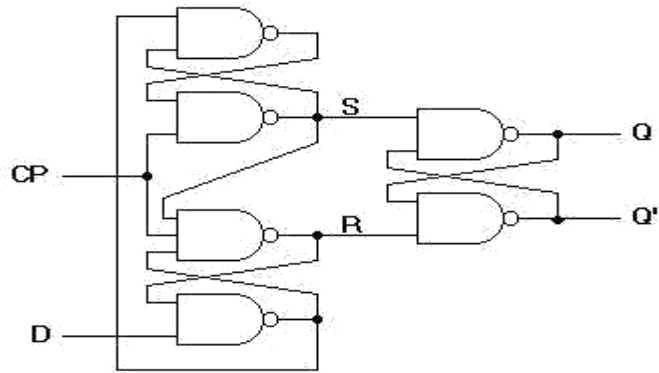
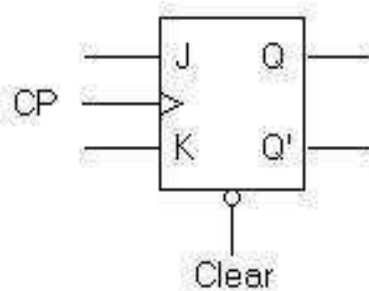


Figure 12. D-type positive-edge triggered flip-flop

When using different types of flip-flops in the same circuit, one must ensure that all flip-flop outputs make their transitions at the same time, i.e., during either the negative edge or the positive edge of the clock pulse.

Direct Inputs

Flip-flops in IC packages sometimes provide special inputs for setting or clearing the flip-flop asynchronously. They are usually called preset and clear. They affect the flip-flop without the need for a clock pulse. These inputs are useful for bringing flip-flops to an initial state before their clocked operation. For example, after power is turned on in a digital system, the states of the flip-flops are indeterminate. Activating the clear input clears all the flip-flops to an initial state of 0. The graphic symbol of a JK flip-flop with an active-low clear is shown in Figure 12.



(a) Graphic Symbol

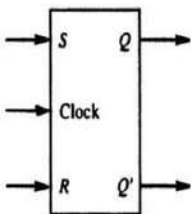
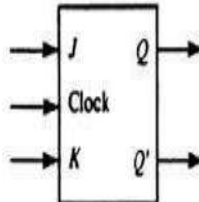
Inputs				Outputs	
Clear	Clock	J	K	Q	Q'
0	x	x	x	0	1
1	0 → 1	0	0	No change	
1	0 → 1	0	1	0	1
1	0 → 1	1	0	1	0
1	0 → 1	1	1	Toggle	

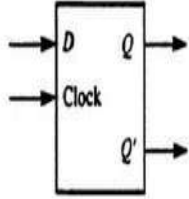
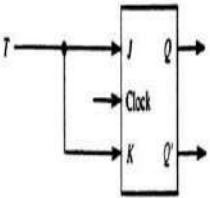
(b) Transition table

Figure 13. JK flip-flop with direct clear

All flip-flops can be divided into four basic types: SR, JK, D and T. They differ in the number of inputs and in the response invoked by different value of input signals. The four types of flip-flops are defined in Table 1.

Table 1. Flip-flop Types

TYPE	FLIP FLOP SYMBOL	CHARACTERISTIC TABLE	CHARACTERISTIC TABLE	EXCITATION TABLE																																			
SR		<table border="1"> <thead> <tr> <th>S</th> <th>R</th> <th>Q(Next)</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Q</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>?</td> </tr> </tbody> </table>	S	R	Q(Next)	0	0	Q	0	1	0	1	0	1	1	1	?	$Q(\text{next}) = S + R'Q$ $SR = 0$	<table border="1"> <thead> <tr> <th>Q</th> <th>Q (Next)</th> <th>S</th> <th>R</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>X</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>X</td> <td>0</td> </tr> </tbody> </table>	Q	Q (Next)	S	R	0	0	0	X	0	1	1	0	1	0	0	1	1	1	X	0
S	R	Q(Next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	?																																					
Q	Q (Next)	S	R																																				
0	0	0	X																																				
0	1	1	0																																				
1	0	0	1																																				
1	1	X	0																																				
JK		<table border="1"> <thead> <tr> <th>J</th> <th>K</th> <th>Q(Next)</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Q</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>Q'</td> </tr> </tbody> </table>	J	K	Q(Next)	0	0	Q	0	1	0	1	0	1	1	1	Q'	$Q(\text{next}) = JQ' + K'Q$	<table border="1"> <thead> <tr> <th>Q</th> <th>Q(Next)</th> <th>J</th> <th>K</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>X</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>X</td> </tr> <tr> <td>1</td> <td>0</td> <td>X</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>X</td> <td>0</td> </tr> </tbody> </table>	Q	Q(Next)	J	K	0	0	0	X	0	1	1	X	1	0	X	1	1	1	X	0
J	K	Q(Next)																																					
0	0	Q																																					
0	1	0																																					
1	0	1																																					
1	1	Q'																																					
Q	Q(Next)	J	K																																				
0	0	0	X																																				
0	1	1	X																																				
1	0	X	1																																				
1	1	X	0																																				
		<table border="1"> <thead> <tr> <th>D</th> <th>Q(Next)</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> </tr> </tbody> </table>	D	Q(Next)			$Q(\text{next}) = D$	<table border="1"> <thead> <tr> <th>Q</th> <th>Q(Next)</th> <th>D</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	Q	Q(Next)	D																												
D	Q(Next)																																						
Q	Q(Next)	D																																					

D		<table border="1" data-bbox="571 192 866 360"> <tr> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> </tr> </table>	0	0	1	1		<table border="1" data-bbox="1225 192 1501 584"> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	0	0	0	0	1	1	1	0	0	1	1	1					
0	0																								
1	1																								
0	0	0																							
0	1	1																							
1	0	0																							
1	1	1																							
T		<table border="1" data-bbox="571 745 866 1070"> <tr> <td>T</td> <td>Q(Next)</td> </tr> <tr> <td>0</td> <td>Q</td> </tr> <tr> <td>1</td> <td>Q'</td> </tr> </table>	T	Q(Next)	0	Q	1	Q'	$Q(\text{next}) = TQ' + T'Q$	<table border="1" data-bbox="1225 701 1501 1240"> <thead> <tr> <th>Q</th> <th>Q(Next)</th> <th>T</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	Q	Q(Next)	T	0	0	0	0	1	1	1	0	1	1	1	0
T	Q(Next)																								
0	Q																								
1	Q'																								
Q	Q(Next)	T																							
0	0	0																							
0	1	1																							
1	0	1																							
1	1	0																							

Lecture 14 & Lecture15

Registers (SISO, SIPO, PIPO, PISO)

Register

A **register** is a sequential circuit with $n + 1$ (not counting the clock) inputs and n output. To each of the outputs corresponds an input. The first n inputs will be called x_0 through x_{n-1} and the last input will be called ld (for load). The n outputs will be called y_0 through y_{n-1} .

When the ld input is 0, the outputs are unaffected by any clock transition. When the ld input is 1, the x inputs are stored in the register at the next clock transition, making the y outputs into copies of the x inputs before the clock transition.

We can explain this behavior more formally with a state table. As an example, let us take a register with $n = 4$. The left side of the state table contains 9 columns, labeled $x_0, x_1, x_2, x_3, ld, y_0, y_1, y_2, y_3$. This means that the state table has 512 rows. We will therefore abbreviate it. Here it is:

ld x3 x2 x1 x0 y3 y2 y1 y0 | y3' y2' y1' y0'

0 --- -- c3 c2 c1 c0 | c3 c2 c1 c0

1 c3 c2 c1 c0 --- -- | c3 c2 c1 c0

As you can see, when ld is 0 (the top half of the table), the right side of the table is a copy of the values of the old outputs, independently of the inputs. When ld is 1, the right side of the table is instead a copy of the values of the inputs, independently of the old values of the outputs. Registers play an important role in computers. Some of them are visible to the programmer, and are used to hold variable values for later use. Some of them are hidden to the programmer, and are used to hold values that are internal to the central processing unit, but nevertheless important.

Shift registers

Shift registers are a type of sequential logic circuit, mainly for storage of digital data. They are a group of flip-flops connected in a chain so that the output from one flip-flop becomes the input of the next flip-flop. Most of the registers possess no characteristic internal sequence of states. All the flip-flops are driven by a common clock, and all are set or reset simultaneously.

In this section, the basic types of shift registers are studied, such as Serial In - Serial Out, Serial In - Parallel Out, Parallel In - Serial Out, Parallel In - Parallel Out, and bidirectional shift registers. A special form of counter - the shift register counter, is also introduced.

Serial In - Serial Out Shift Registers (SISO)

A basic four-bit shift register can be constructed using four D flip-flops, as shown below. The operation of the circuit is as follows. The register is first cleared, forcing all four outputs to zero. The input data is then applied sequentially to the D input of the first flip-flop on the left (FF0). During each clock pulse, one bit is transmitted from left to right. Assume a data word to be 1001. The least significant bit of the data has to be shifted through the register from FF0 to FF3.

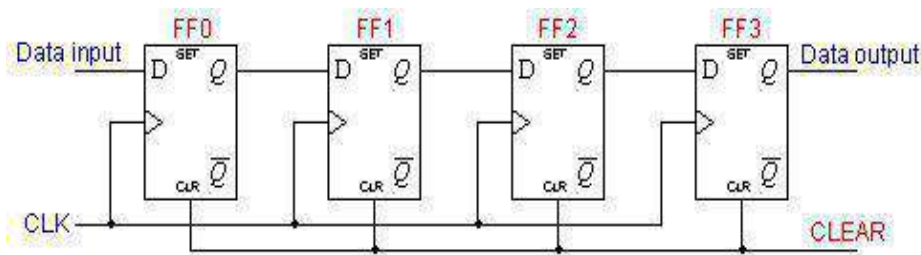


Figure 1. Serial In - Serial Out Shift Registers Graphic Diagram

In order to get the data out of the register, they must be shifted out serially. This can be

done destructively or non-destructively. For destructive readout, the original data is lost and at the end of the read cycle, all flip-flops are reset to zero. To avoid the loss of data, an arrangement for a non-destructive reading can be done by adding two AND gates, an OR gate and an inverter to the system. The construction of this circuit is shown below.

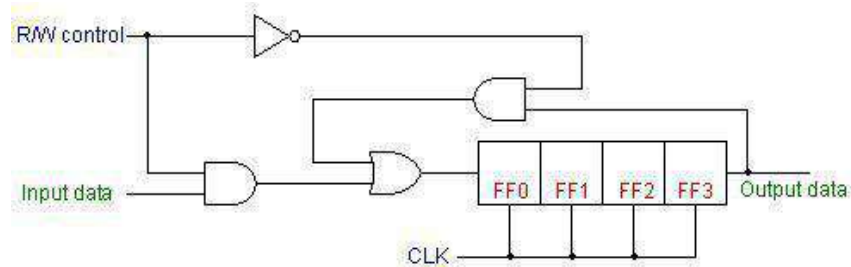


Figure 2. SISO Shift Register Logic Diagram

Serial In - Parallel Out Shift Registers (SIPO)

For this kind of register, data bits are entered serially in the same manner as discussed in the last section. The difference is the way in which the data bits are taken out of the register. Once the data are stored, each bit appears on its respective output line, and all bits are available simultaneously. A construction of a four-bit serial in - parallel out register is shown below.

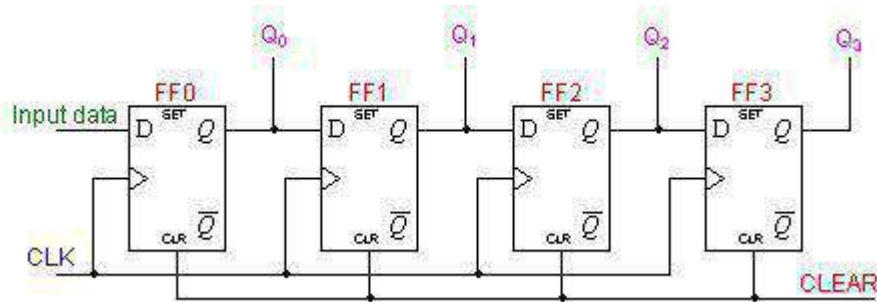


Figure 3. Serial In - Parallel Out Shift Registers Graphic Diagram

A four-bit parallel in - serial out shift register is shown below. The circuit uses D flip-flops and NAND gates for entering data (ie writing) to the register.

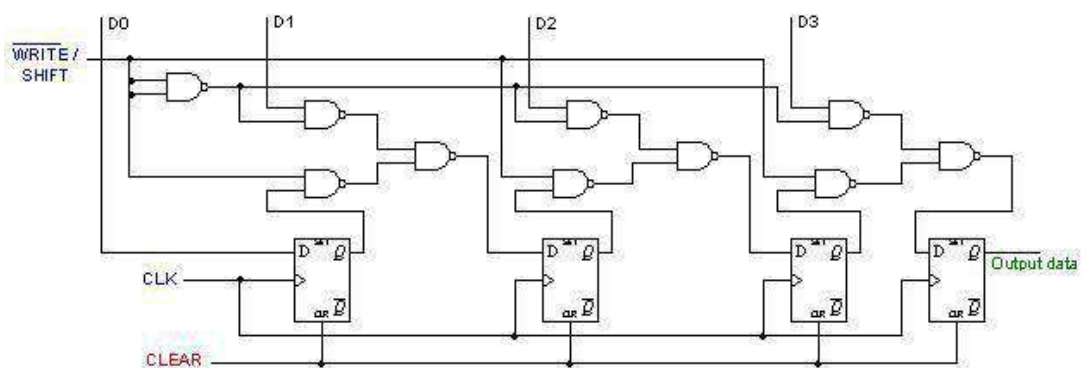


Figure 4. Serial In - Parallel Out Shift Registers Logic Diagram

D0, D1, D2 and D3 are the parallel inputs, where D0 is the most significant bit and D3 is the least significant bit. To write data in, the mode control line is taken to LOW and the data is clocked in. The data can be shifted when the mode control line is HIGH as SHIFT is active high

Parallel In - Parallel Out Shift Registers (PIPO)

For parallel in - parallel out shift registers, all data bits appear on the parallel outputs immediately following the simultaneous entry of the data bits. The following circuit is a four-bit parallel in - parallel out shift register constructed by D flip-flops.

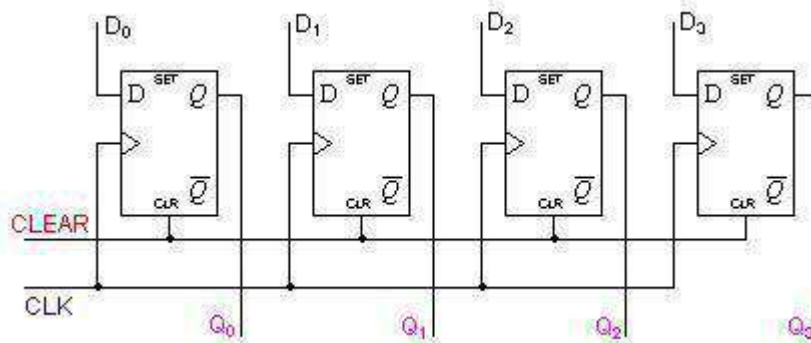


Figure 5. Parallel In - Parallel Out Shift Registers Graphic Diagram

The D's are the parallel inputs and the Q's are the parallel outputs. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously.

Bidirectional Shift Registers

The registers discussed so far involved only right shift operations. Each right shift operation has the effect of successively dividing the binary number by two. If the operation is reversed (left shift), this has the effect of multiplying the number by two. With suitable gating arrangement a serial shift register can perform both operations.

A *bidirectional*, or *reversible*, shift register is one in which the data can be shift either left or right. A four-bit bidirectional shift register using D flip-flops is shown below.

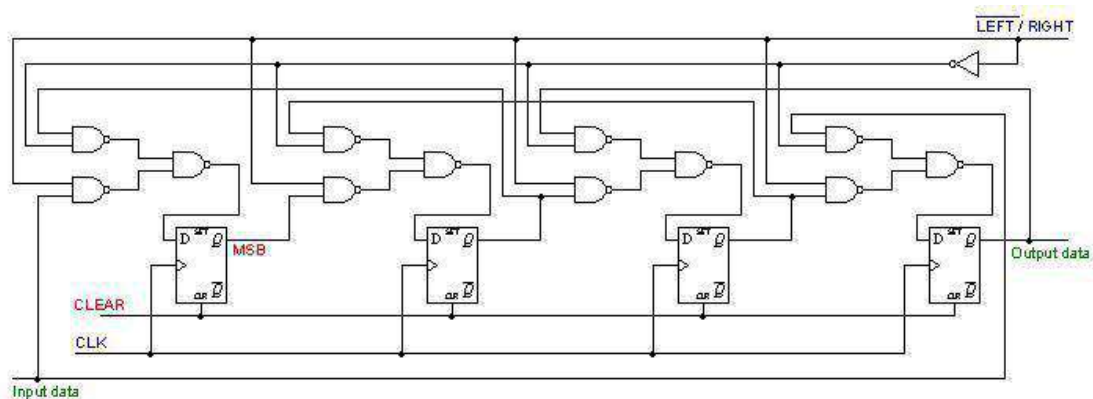


Figure 6. Logic Diagram of Bidirectional Shift Register

Here a set of NAND gates are configured as OR gates to select data inputs from the right or left adjacent bi stables, as selected by the LEFT/RIGHT control line.

Lecture 16

Ring counter, Johnson counter

Counters

A sequential circuit that goes through a prescribed sequence of states upon the application of input pulses is called a **counter**. The input pulses, called **count pulses**, may be clock pulses. In a counter, the sequence of states may follow a binary count or any other sequence of states. Counters are found in almost all equipment containing digital logic. They are used for counting the number of occurrences of an even and are useful for generating timing sequences to control operations in a digital system.

A *counter* is a sequential circuit with 0 inputs and n outputs. Thus, the value after the clock transition depends only on old values of the outputs. For a counter, the values of the outputs are interpreted as a sequence of *binary digits* (see the section on binary arithmetic).

We shall call the outputs o_0, o_1, \dots, o_{n-1} . The value of the outputs for the counter after a clock transition is a binary number which is *one plus* the binary number of the outputs before the clock transition.

We can explain this behavior more formally with a state table. As an example, let us take a counter with $n = 4$. The left side of the state table contains 4 columns, labeled $o_0, o_1, o_2,$ and o_3 . This means that the state table has 16 rows. Here it is in full:

$o_3 \ o_2 \ o_1 \ o_0 \ | \ o_3' \ o_2' \ o_1' \ o_0'$

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

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

Figure 1. State Table

Again, the apparent disadvantage of this counter is that the maximum available states are not fully utilized. Only eight of the sixteen states are being used. The right hand side of the table is always one plus the value of the left hand side of the table, except for the last line, where the value is 0 for all the outputs. We say that the counter *wraps around*.

Important variations include:

11. The ability to count up or down according to the value of an additional input
12. The ability to count or not according the the value of an additional input
13. The ability to clear the contents of the counter if some additional input is 1
14. The ability to act as a register as well, so that a predetermined value is loaded when some additional input is 1
15. The ability to count using a different representation of numbers from the normal (such as Gray-codes, 7-segment codes, etc)
16. The ability to count with different increments that 1

Shift Register Counters

Two of the most common types of shift register counters are introduced here: ***the Ring counter*** and ***the Johnson counter***. They are basically shift registers with the serial outputs connected back to the serial inputs in order to produce particular sequences. These registers are classified as counters because they exhibit a specified sequence of states.

Ring Counters

A ring counter is basically a circulating shift register in which the output of the most significant stage is fed back to the input of the least significant stage. The following is a 4-bit ring counter constructed from D flip-flops. The output of each stage is shifted into the next stage on the positive edge of a clock pulse. If the CLEAR signal is high, all the flip-flops except the first one FF0 are reset to 0. FF0 is preset to 1 instead.

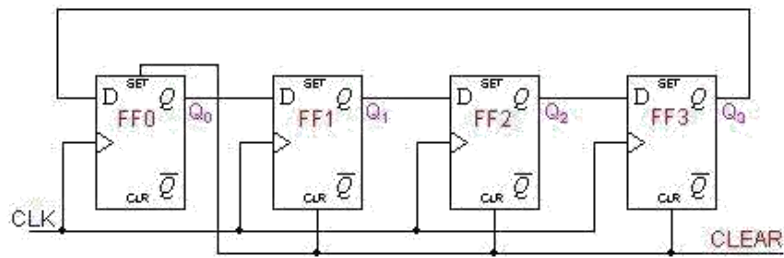


Figure 2. Graphic Diagram of Ring Counter

Since the count sequence has 4 distinct states, the counter can be considered as a mod-4 counter. Only 4 of the maximum 16 states are used, making ring counters very inefficient in terms of state usage. But the major advantage of a ring counter over a binary counter is that it is self-decoding. No extra decoding circuit is needed to determine what state the counter is in.

CLOCK PULSE	Q3	Q2	Q1	Q0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0

Figure 3. State Sequence Of Ring Counter

Johnson Counters

Johnson counters are a variation of standard ring counters, with the inverted output of the last stage fed back to the input of the first stage. They are also known as twisted ring counters. An n -stage Johnson counter yields a count sequence of length $2n$, so it may be considered to be a mod- $2n$ counter. The circuit above shows a 4-bit Johnson counter. The state sequence for the counter is given in the table

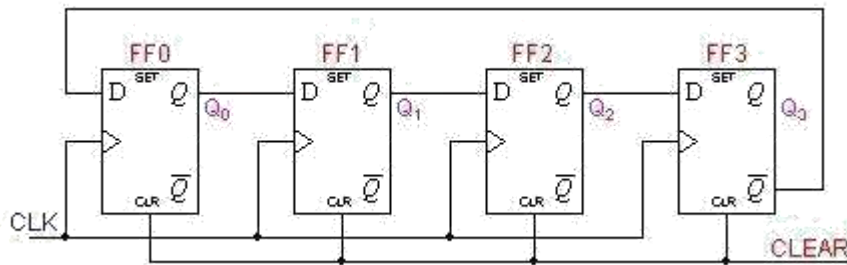


Figure 4. Graphic Diagram Of Johnson Counter

Clock Pulse	Q3	Q2	Q1	Q0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	1
3	0	1	1	1
4	1	1	1	1
5	1	1	1	0
6	1	1	0	0
7	1	0	0	0

Figure 4. State Sequence Of Johnson Counter

Lecture 17

Basic concept of Synchronous and Asynchronous counters

Synchronous Counters

All flip-flops are clocked simultaneously by an external clock. Means clock input of all flip flops are connected to same external clock. Synchronous counters are faster than asynchronous counters because of the simultaneous clocking. Synchronous counters are an example of *state machine* design because they have a set of states and a set of transition rules for moving between those states after each clocked event.

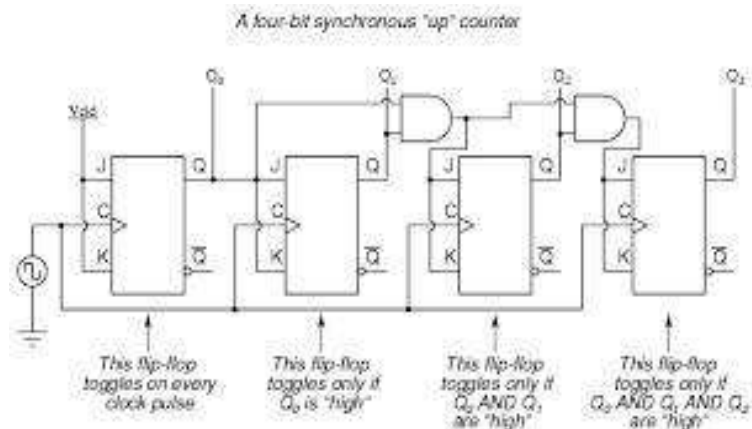


Figure 1. Synchronous Counter

Asynchronous Counters

Only the first flip-flop is clocked by an external clock. All subsequent flip-flops are clocked by the output of the preceding flip-flop. means output of previous flip-flop is

connected to clock input of next flip flop. Asynchronous counters are slower than synchronous counters because of the delay in the transmission of the pulses from flip-flop to flip-flop. Asynchronous counters are also **called ripple-counters** because of the way the clock pulse ripples it way through the flip-flops.

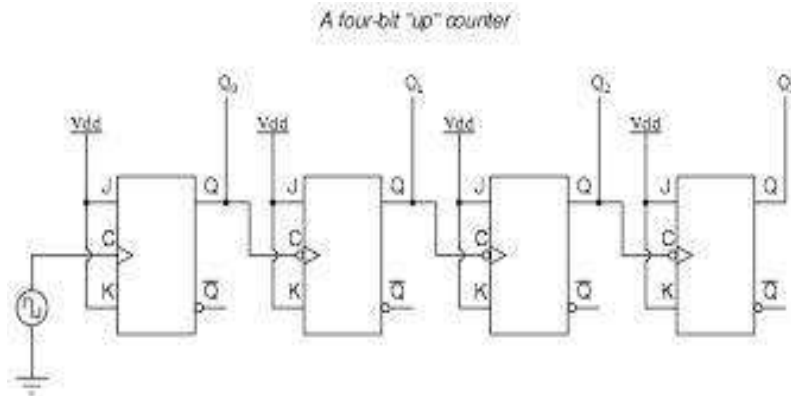


Figure 2. Asynchronous Counter

Asynchronous Counter/Ripple counters

- can be constructed using several flip flops
- consider the following arrangement
- with $J = K = 1$ each flip flop toggles on the falling edge of its clock input.

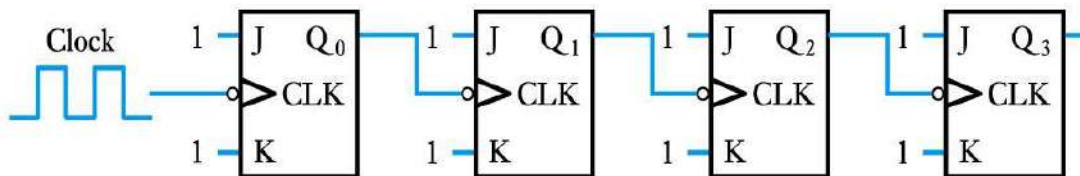


Figure 3. Graphic Diagram of asynchronous counter

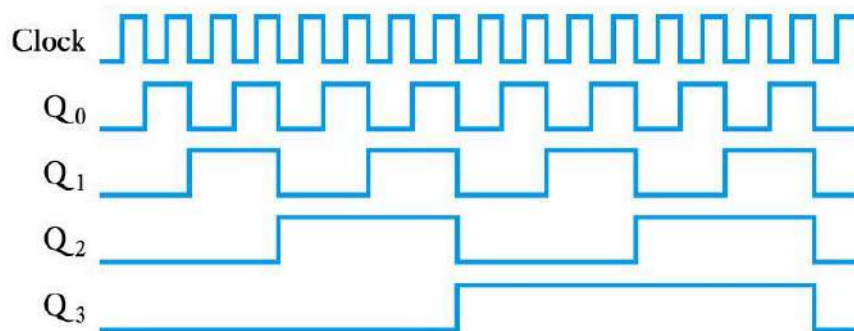


Figure 4. Clock Pulse

Each stage toggles at half the frequency of the previous stage

- acts as a frequency divider
- divides frequency by ‘2 to the power n’(n is the number of stages)

Lecture 18

Design of Mod N Counter

Modulo-*N* counters

- by using an appropriate number of stages the earlier counter can count modulo any power of 2
- to count to any other base we add reset circuitry
- e.g. the modulo-10 or decade counter shown here

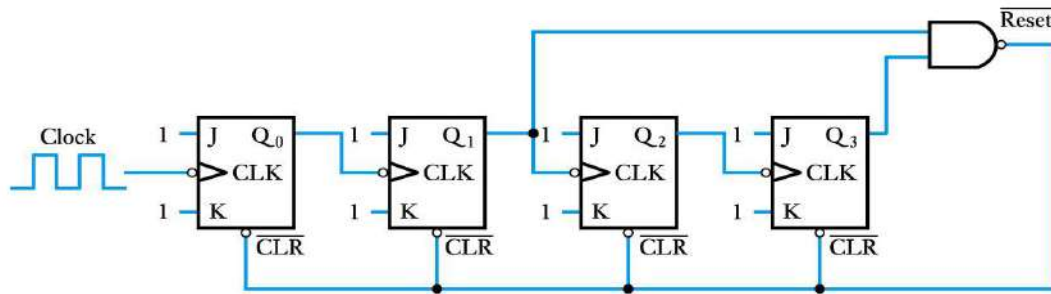


Figure 1. Graphic Diagram of Mod N Counter

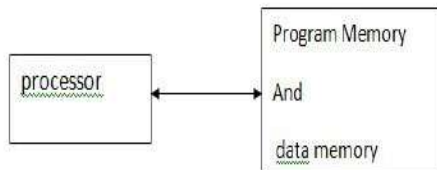
Lecture19:

Stored program concept

A **stored-program computer** is one that stores [program instructions](#) in electronic memory. Two types of stored program architecture is there- Von Neumann architecture and Harvard Architecture

Von Neumann Architecture:

It is named after the mathematician John Von Neumann introduced this stored program concept at Institute of Advanced Studies(IAS),Princeton .It is also called Princeton Architecture.The computer has single storage system(memory) for storing data as well as program to be executed. Instructions are executed sequentially one at a time. Here, instructions and data cannot be fetched simultaneously as it uses same control signal and same bus for instruction and data fetching.



Processor needs two clock cycles to complete an instruction. Pipelining the instructions is not possible with this architecture. In the first clock cycle the processor gets the instruction from memory and decodes it. In the next clock cycle the required data is taken from memory. For each instruction this cycle repeats and hence needs two

cycles to complete an instruction. This is a relatively older architecture and was replaced by Harvard architecture.

Harvard Architecture:

Howard Aiken of Harvard University introduced a different stored program architecture which uses two separate memories for storing data and program. It also uses separate buses and signals for instruction and data. Processor can complete an instruction in one cycle if appropriate pipelining strategies are implemented. In the first stage of pipeline the instruction to be executed can be taken from program memory. In the second stage of pipeline data is taken from the data memory using the



decoded instruction or address. Most of the modern computing architectures are based on Harvard architecture.

Lecture20: Introduction to CPU:

Definition:

The **central processing unit (CPU)** of a computer is a piece of hardware that is the **brain** of the computer. It is the electronic circuitry within a computer that carries out the instructions of a computer program. It performs the basic arithmetical, logical, and input/output operations of a computer system. Traditionally, the term "CPU" refers to a processor, more specifically to its processing unit and control unit (CU).

Components:

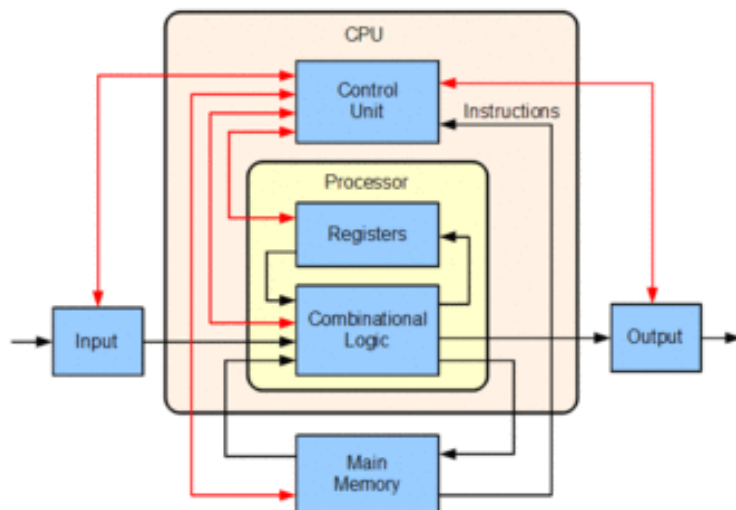
Principal components of a CPU include-

Arithmetic logic unit (ALU) that performs arithmetic and logic operations.

Second is the control unit (CU), which manages the various components of the computer. It reads and interprets instructions from memory and transforms them into a series of signals to activate other parts of the computer. The control unit calls upon the arithmetic logic unit to perform the necessary calculations.

Third is processor registers that supply operands to the ALU and store the results of ALU operations.

CPUs are located on the motherboard. Most modern CPUs are microprocessors, meaning they are contained on a single integrated circuit (IC) chip. An IC that contains a CPU may also contain memory, peripheral interfaces, and other components of a computer;



Lecture21:

Introduction to ALU:

An arithmetic logic unit (ALU) is a combinational digital electronic circuit that performs arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers.

How an ALU Works

An ALU performs a number of basic arithmetic and bitwise logic functions. They include

Arithmetic operations

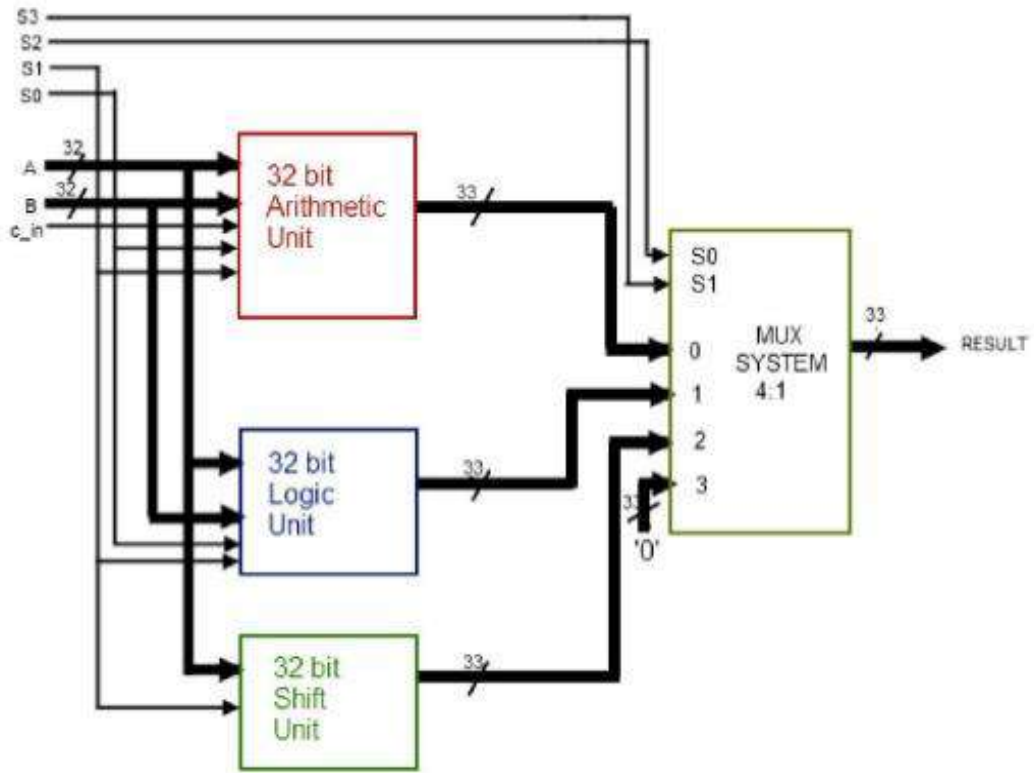
- *Add*: A and B are summed and the sum appears at Y and carry-out.
- *Add with carry*: A, B and carry-in are summed and the sum appears at Y and carry-out.
- *Subtract*: B is subtracted from A (or vice versa) and the difference appears at Y and carry-out. For this function, carry-out is effectively a "borrow" indicator. This operation may also be used to compare the magnitudes of A and B; in such cases the Y output may be ignored by the processor, which is only interested in the status bits (particularly zero and negative) that result from the operation.
- *Subtract with borrow*: B is subtracted from A (or vice versa) with borrow (carry-in) and the difference appears at Y and carry-out (borrow out).
- *Two's complement (negate)*: A (or B) is subtracted from zero and the difference appears at Y.
- *Increment*: A (or B) is increased by one and the resulting value appears at Y.
- *Decrement*: A (or B) is decreased by one and the resulting value appears at Y.
- *Pass through*: all bits of A (or B) appear unmodified at Y. This operation is typically used to determine the parity of the operand or whether it is zero or negative.

Bitwise logical operations

- *AND*: the bitwise AND of A and B appears at Y.
- *OR*: the bitwise OR of A and B appears at Y.
- *Exclusive-OR*: the bitwise XOR of A and B appears at Y.
- *Ones' complement*: all bits of A (or B) are inverted and appear at Y.

Bit shift operations

ALU shift operations cause operand A (or B) to shift left or right (depending on the opcode) and the shifted operand appears at Y. Simple ALUs typically can shift the operand by only one bit position



32-bit Arithmetic Logic Unit

FUNCTIONS OF ALU

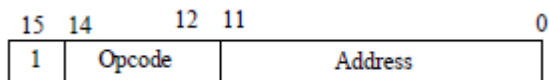
S3	S2	S1	S0	Cin	Result	Operation
0	0	0	0	0	A + B	Add
0	0	0	0	1	A + B + 1	Add with Carry
0	0	0	1	0	A + (not B)	Subtract with Borrow
0	0	0	1	1	A + (not B) + 1	Subtract
0	0	1	0	0	A	Transfer A
0	0	1	0	1	A + 1	Increment A
0	0	1	1	0	A - 1	Decrement A
0	0	1	1	1	A	Transfer A
0	1	0	0	X	A and B	And
0	1	0	1	X	A or B	Or
0	1	1	0	X	A xor B	Xor
0	1	1	1	X	(Not A)	Complement
1	0	0	X	X	LSR A	Right Shift
1	0	1	X	X	LSL A	Left Shift

LECTURE 22

INSTRUCTION FORMATS

A computer instruction is a binary code that specifies a sequence of micro operations for the computer. The bits of the instruction are divided into groups called fields. The most common fields found in instruction format are: -

1. An operation code field that specifies the operation to be performed
2. An address field that designates a memory address or a processor register where the operand is stored
3. A mode field that specifies the way in which the operand address is determined



Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Three types of CPU organizations are-

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

In Single accumulator organization, one address instruction is used as instruction uses only one operand address field. Here accumulator is used implicitly for data manipulation.

LOAD A $AC \leftarrow M[A]$
ADD B $AC \leftarrow A[C] + M[B]$

In General register organization, two or three address instructions are used as there are multiple registers.

ADD R1, B $R1 \leftarrow R1 + M[B]$
ADD R1, A, B $R1 \leftarrow M[A] + M[B]$.

A stack-organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.

PUSH A $TOS \leftarrow A$
PUSH B $TOS \leftarrow B$
ADD $TOS \leftarrow (A + B)$

INSTRUCTION CYCLE:

A program residing in the memory unit of the computer consists of a sequence of instructions.

The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into following sequence of sub cycles or phases-

1. Fetch an instruction from memory.
2. Decode the instruction

3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

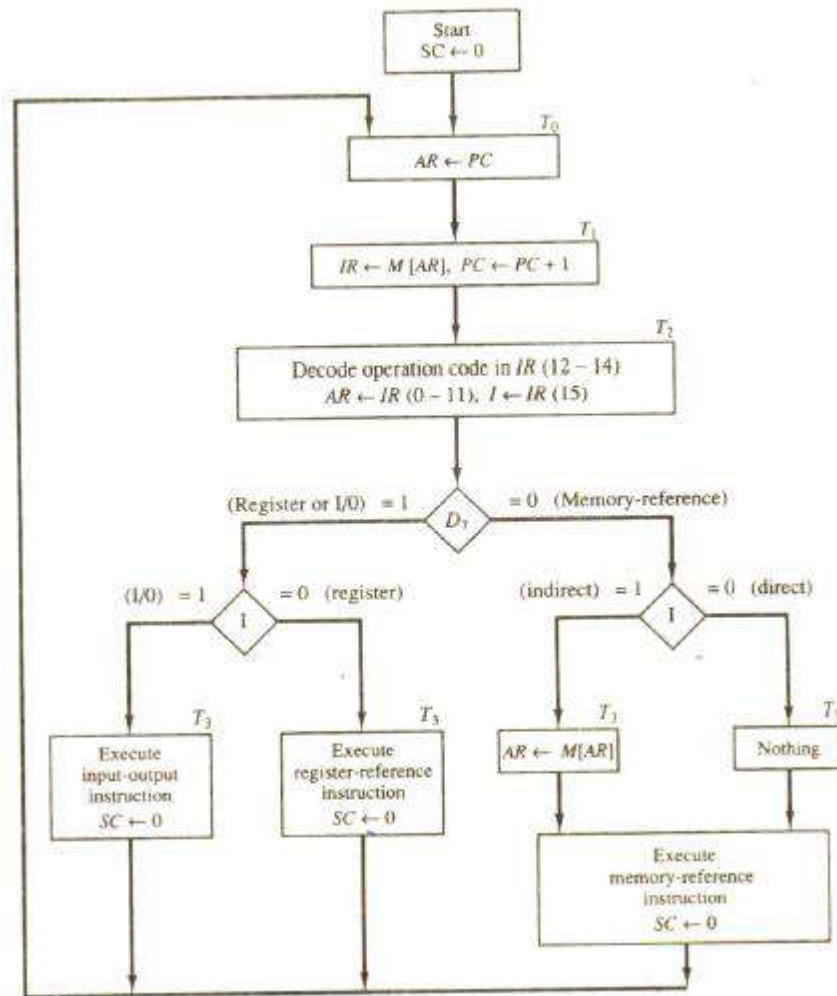
Upon the completion of step 4, the control goes back to step 1 and continues for next instruction. This process continues indefinitely unless a HALT instruction occurs.

Initially, the program counter PC holds the address of the first instruction in the program. Content of PC is copied into AR (Address Register- it holds the address of memory that is to be read or written). Instruction is fetched from address in AR and stored into IR (Instruction Register). Now PC is incremented to next instruction address. Control unit decodes the current instruction and it is executed in CPU.

$AR \leftarrow PC$

$IR \leftarrow M[AR], PC \leftarrow PC + 1$

$D_0, \dots, D_1 \leftarrow \text{Decode } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$



Flowchart for instruction cycle

LECTURE 23

ADDRESSING MODES:

An addressing mode specifies how to calculate the effective address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere. Effective address is the actual address of the operand.

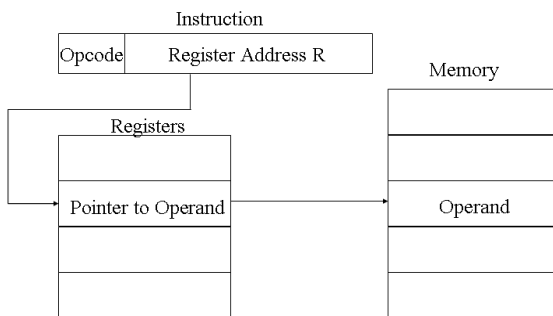
Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “**complement accumulator**” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.

Immediate Mode: In this mode the actual operand is explicitly specified in the instruction.e.g. **ADI 05**

Register Mode: In this mode the operands are in processor registers. Here, the address field specifies a processor register.

ADD R1, R2

Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.

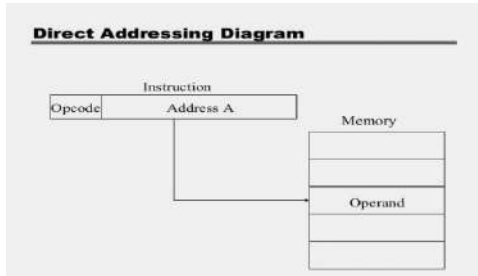


MOV A, (R2)
Content of register R2 is memory address of the operand.

Auto increment or Auto decrement Mode: This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

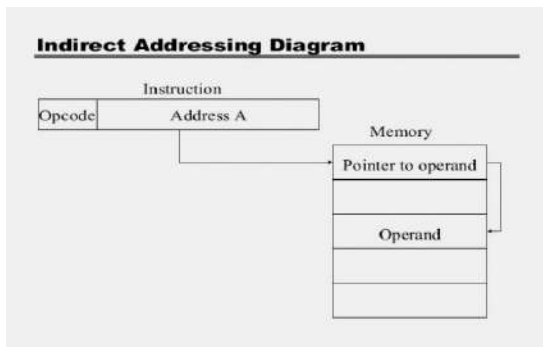
In first case, after accessing operand, content of the register is automatically incremented to point to the next item in a list. In the latter one, content of register are decremented first and then used as effective address of operand. These two modes are useful for accessing operands from successive memory locations.

Direct Address Mode: In this mode the operand resides in memory and its address is given directly by the address field of the instruction.



STA 2500

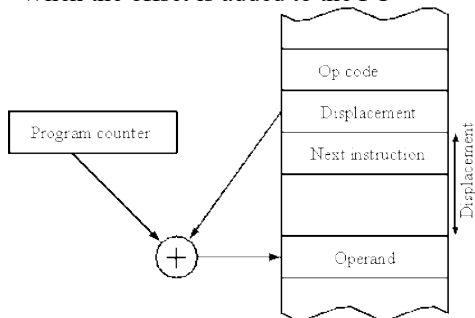
Indirect Address Mode: In this mode the address field of the instruction gives the memory address which holds the actual address of operand.



MOV R1,(5000)

Relative Addressing mode: In relative addressing mode, contents of Program Counter PC is added to address part of instruction to obtain effective address. The address part of the instruction is called as offset and it can +ve or -ve.

- When the offset is added to the PC



Indexed Addressing mode: The address of the operand is obtained by adding to the contents of the index register to the address part of instruction. Index register XR is a special processor register that contain the index value of current operand. This mode is used to access an array whose elements are in successive memory locations. The address field of the instruction represents the starting address of the array. By incrementing or decrementing index register different element of the array can be accessed.

$$\text{OPCODE X} \quad \text{Effective Address} = X + [\text{XR}]$$

Base Register Addressing Mode: In this mode the content of a base register (BR) is added to the address part of the instruction to obtain the effective address. The base register holds the beginning/base address and the address

part of instruction holds the offset/displacement. This mode is used in computers to facilitate the relocation of programs in memory. During relocation, BR is set to new address but offset remains same.

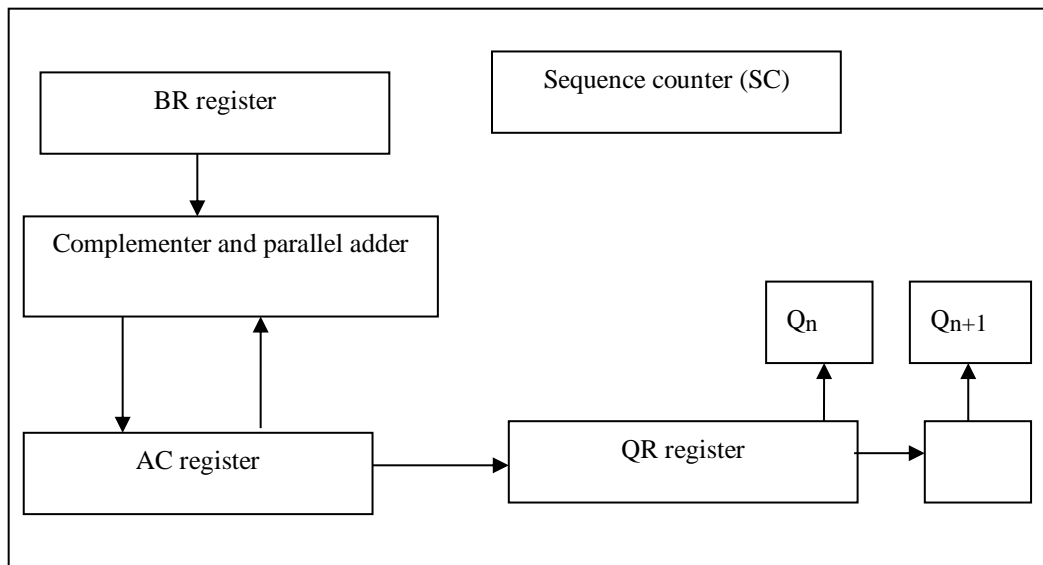
Lecture 24

Booth's algorithm

Booth's multiplication algorithm is a [multiplication algorithm](#) that multiplies two signed [binary](#) numbers in [two's complement notation](#). The [algorithm](#) was invented by [Andrew Donald Booth](#) in 1950 while doing research on [crystallography](#) at [Birkbeck College](#) in [Bloomsbury, London](#). Booth used desk calculators that were faster at [shifting](#) than adding and created the algorithm to increase their speed.

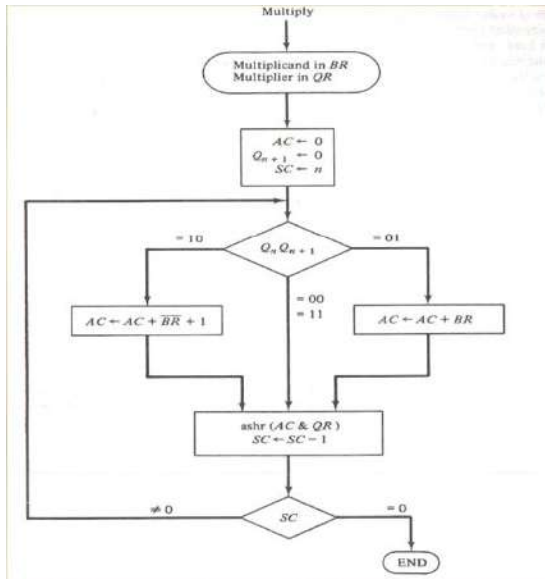
Hardware Design:

The hardware implementation of Booth algorithm is shown in fig 1. In this design, registers are used for different purposes. BR register holds the multiplicand and QR register holds the multiplier. Q_n signifies the least significant bit of the multiplier in QR register. AC register is added to hold the intermediate and final results. A flip-flop, Q_{n+1} is added at the end of the multiplier to facilitate a double bit inspection of the multiplier. Complementer and parallel adder is added to receive the input from BR and AC register and also provides the output to the AC register. Extra flip-flop sequence counter(SC) is added to hold the number of bits present in the multiplier.



The flowchart for Booth algorithm

The flowchart for Booth algorithm is shown in Fig. 2. We have assumed that AC register and Q_{n+1} are set to value zero. SC holds the number of bits present in the multiplier (QR). By checking the bits present in Q_n and Q_{n+1} , either addition or subtraction operation is followed. Operation is continued until SC is zero.



Case 1:

$Q_n=0$

$Q_{n+1}=0$

Right shift both AC and QR

Decrease the content of SC by 1

Case 2:

$Q_n=0$

$Q_{n+1}=1$

$AC \leftarrow AC + BR$

Right shift both AC and QR

Decrease the content of SC by 1

Case 3:

$Q_n=1$

$Q_{n+1}=0$

$AC \leftarrow AC + BR (1\text{'s complement}) + 1$

Right shift both AC and QR

Decrease the content of SC by 1

Case 4:

$Q_n=1$

$Q_{n+1}=1$

Right shift both AC and QR

Decrease the content of SC by 1

Example

Example of Booth's algorithm for binary multiplication.

$$7 \times (-3) = ?$$

$$7 = 0111 \text{ (BR) - Multiplicand}$$

$$-3 = 1101 \text{ (QR) - Multiplier}$$

Table.

Q_n	Q_{n+1}	BR=0111 $\overline{\text{BR}+1}=1001$	Ac	QR	Q_{n+1}	Sc
1	0	Initial	0000	1101	0	100
		Subtract 1001				
		Right shift	1100	1101	1	011
0	1	Addition	0111			
		0011				
		[carry is to be discarded in 2's complement arithmetic]				
		Right shift	0001	1111	0	010
1	0	Subtract	1001			
		Right shift	1010	1111		
		Right shift	1101	0111	1	001
1	1	Right shift	1110	1011	1	000
		Answer				

$$\therefore 7 \times (-3) = -21$$

$$= 11101011 \text{ (Ans)}$$

Lecture 25
Restoring Division Algorithm
Hardware Design

In restoring division, the divisor is shift-positioned and subtracted from the dividend. If subtraction produces a negative result at any bit position relative to the dividend, the operation at that bit position is unsuccessful, and a 0 is placed in the corresponding location of the quotient. The divisor is added back (restored) to the result of the division operation, then the next highest bit of the dividend is shifted into the left bit position of the result. As each bit of the dividend is shifted from right to left, the quotient is built up from left to right. After n shifts, where n represents the number of bits in the dividend, the division operation is complete. Complete hardware for restoring division is shown in Fig.1. In this figure an n -bit positive divisor is loaded into register M and n -bit dividend is loaded into register Q at the start of the operation. After the division is complete, the n -bit quotient is in register Q and the remainder is in register A . The result after the last restore operation is the remainder.

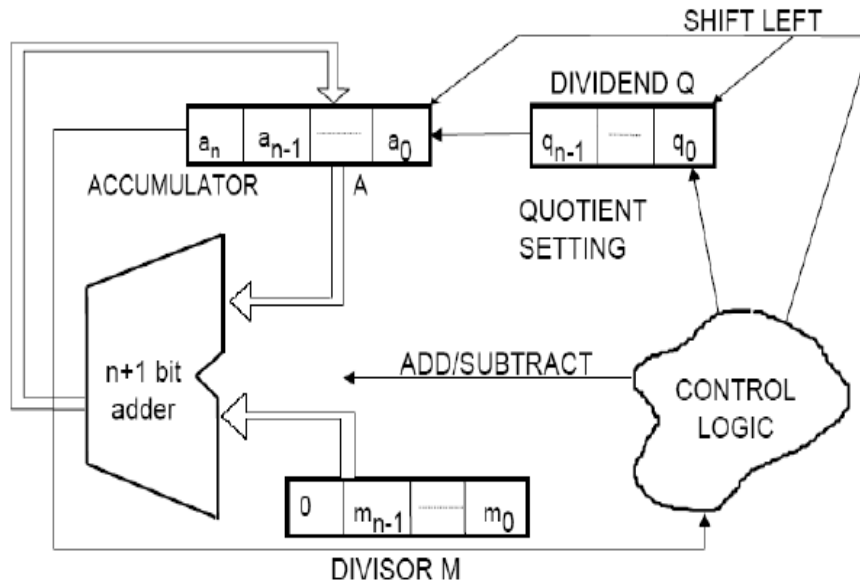
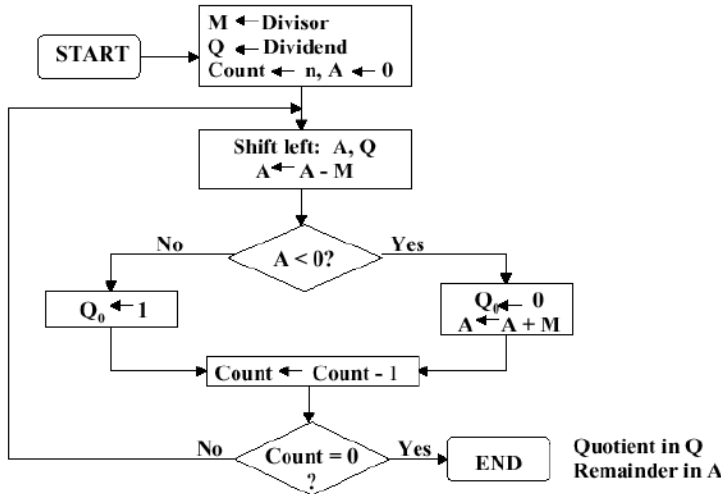


Fig.1

Flowchart:

Figure below shows the flowchart for restoring algorithm.



The steps are as follows:-

Step 1: Set count to n and A to 0.

Step 2: Set Multiplicand to M and Multiplier to Q

Step 3: Shift A and Q left by one binary position

Step 4: Subtract divisor M from A and place the answer in A ($A \leftarrow A - M$)

Step 5: If the sign bit of A is 1, set q_0 to 0 and add divisor back to A (restore A)

Step 6, otherwise set q_0 to 1

Step 7: Repeat steps 3 to 6 up to count 0.

Example

Restoring Division Algorithm

(10) 11) 1000(0)

11

10 - Remainder

A	Q	M	
Initialy 0000	1000	00011	} 1st cycle
LS 00001	000-		
Subtract 11101	0000		
Set q_0 01110			
Restore 11			} 2nd cycle
00001			
LS 00010	000-		
Subtract 11101	0000		
Set q_0 01111			} 3rd cycle
Restore 11			
00010	0001		
LS 00100	000-		
Subtract 11101	0001		} 4th cycle
Set q_0 00001			
Restore 11			
00010	001-		
Subtract 11101	0010		
Set q_0 01111			
Restore 11			
00010	0010		
Remainder	Quotient		

Non-restoring division algorithm

Restoring division can be improved using non-restoring algorithm. The effect of restoring algorithm actually is:

If A is positive, we shift it left and subtract M that is compute $2A-M$.

If A is negative, we restore it $(A+M)$, shift it left, and subtract M, that is, $2(A+M) - M = 2A+M$. Set q_0 to 1 or 0 appropriately.

Algorithm is shown below

Step 1: Set A to 0.

Step 2: Repeat n times:

Step 2.1: If the sign of A is positive:

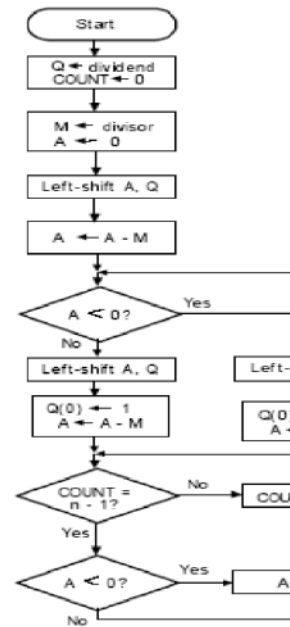
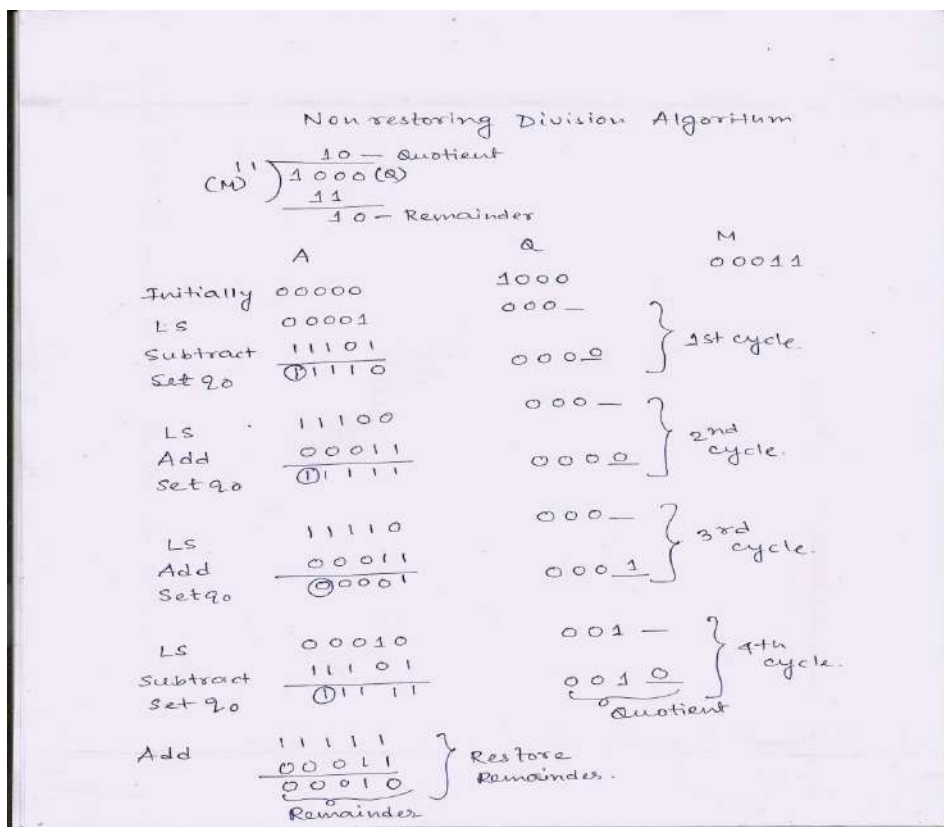
Shift A and Q left and subtract M. Set q_0 to 1.

Step 2.2: Else if the sign of A is negative:

Shift A and Q left and add M. Set q_0 to 0.

Step 3: If the sign of A is 1, add A to M.

Flowchart of non-restoring algorithm is shown below.



Example

Lecture 26

Floating point Representation

Any floating point number has two components. Components are sign, magnitude and exponent. Floating point values are generally represented with a normalized scientific notation. Normalized scientific notation represents a number consisting of single non-zero digit to the left of the decimal (binary) point.

For Example: $12.258 = 1.2258 \times 10^1$. In this example, 0.2258 is the mantissa and 1 is the exponent part.

A standard notation facilitates for easy exchange of data between machines and also reduces complexity of hardware algorithms. With floating point representation, we can represent from a large to small number easily with maximizing precision (the number of digits) at both ends of the scale.

The IEEE 754 standard defines the representation of 32 bits single precision and 64 bits double precision of floating point numbers.

IEEE 32 bits Floating-Point Format

Sign(S)	Exponent(E')	1. Mantissa(M)
1 bit	8 bits	23 bits
0 for + 1 for -	Signed exponent in excess -127 form	Hidden 1 (not stored)
32 bits		

Single Precision Representation: $\pm 1.M \times 2^{E' - 127}$

In single precision format original signed exponent E and base 2 are not stored in register. The value in the exponent part means biased exponent and calculated as $E' = E + 127$. This format is known as excess -127 format. E' lies in the range $0 \leq E' \leq 255$. The end values 0 and 255 represent special values. The actual exponent (E) is in the range $-126 \leq E \leq 127$.

Special Case

Case 1: $E' = 0$

If the mantissa $M = 0$, the exact 0 value is represented.

If the mantissa $M \neq 0$, denormal number is represented. The denormal number introduces the concept of underflow.

Case 2: $E' = 255$

If the mantissa $M = 0$, it represents infinity (∞).

If the mantissa $M \neq 0$, it represents Not a Number (NaN).

IEEE 64 bits Double Precision Format

Sign(S)	Exponent(E')	1. Mantissa(M)
1 bit	11 bits	52 bits
0 for +	Signed exponent in excess	Hidden 1 (not stored)

1 for -	-1023 form	
64 bits		

Single Precision Representation: $\pm 1.M \times 2^{E' - 1023}$

In double precision format, the end values 0 and 2047(biased exponent) represent special values.

Examples

Q1. Suppose that IEEE-754 32-bit floating-point representation pattern is 0 10000000 110 0000 0000 0000 0000 0000.

Sign bit $S = 0 \Rightarrow$ positive number

$E = 1000\ 0000 = 128$ (in normalized form)

Fraction is 1.11B (with an implicit leading 1) $= 1 + 1 \times 2^{-1} + 1 \times 2^{-2} = 1.75$

The number is $+1.75 \times 2^{(128-127)} = +3.5$

Q2. Suppose that IEEE-754 32-bit floating-point representation pattern is 1 01111110 100 0000 0000 0000 0000 0000

Sign bit $S = 1 \Rightarrow$ negative number

$E = 0111\ 1110 = 126$ (in normalized form)

Fraction is 1.1 (with an implicit leading 1) $= 1 + 2^{-1} = 1.5$

The number is $-1.5 \times 2^{(126-127)} = -0.75$

Lecture 27

Floating point operation

Arithmetic operations on floating point numbers consist of addition, subtraction, multiplication and division.

Steps for Addition/Subtraction

1. Choose the number with smaller exponent and shift its mantissa right a number of positions equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissa and determine the sign of the result.
4. Normalize the result, if necessary.

Steps for Multiplication (based on IEEE Single precision format)

1. Add the exponent and subtract 127.
2. Add the exponent and subtract 127.
3. Normalize the result, if necessary

Steps for Division (based on IEEE Single precision format)

1. Subtract the exponent and add 127.
2. Divide the mantissa and determine the sign of the result.
3. Normalize the result, if necessary

Rounding

Arithmetic operations on fl. pt. values compute results that cannot be represented in the given amount of precision. So, we must round results. There are three ways of rounding techniques.

Round toward 0

This method is also known as truncation.

Example:

0.5683 if 3 decimal places available, 0.568
 if 2 decimal places available, 0.57

Round toward + infinity

Example:

1.46 if 2 decimal places, 1.5

Round toward - infinity

Example:

-2.87 if 2 decimal places, -2.9

Examples of floating point operation

Example.

Floating point addition.

a) $12_{(10)} + 1.25_{(10)} = ?$ (Using 14 bit floating point model)

$$12_{10} = 0.1100 \times 2^4$$

$$1.25_{10} = 0.101 \times 2^1$$

$$= 0.000101 \times 2^4$$

$$\begin{array}{r} 0 \quad 10100 \quad 11000000 \\ + 0 \quad 10100 \quad 00010100 \\ \hline 0 \quad 10100 \quad 11010100 \end{array}$$

Ans. 0.110101×2^4

b) $12_{10} \times 1.25_{10} = ?$ (Using 14 bit floating point model)

$$12_{10} = 0.1100 \times 2^4$$

$$1.25_{10} = 0.101 \times 2^1$$

$$= 0.000101 \times 2^4$$

$$\begin{array}{r} 0 \quad 10100 \quad 11000000 \\ \times 0 \quad 10001 \quad 10100000 \\ \hline 0 \quad 10101 \quad 01111000 \end{array}$$

Ans. $0.0111100 \times 2^5 = 0.1111 \times 2^4$

\therefore Normalized product requires an exponent of $22_{10} = 10110_2$

LECTURE 28 :

RAM:

Random-access memory (RAM) is a form of computer data storage which stores frequently used program instructions to increase the general speed of a system. A random-access memory device allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. In contrast, with other direct-access data storage media such as hard disks, CD-RWs, DVD-RWs and the older drum memory, the time required to read and write data items varies significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement. In today's technology, random-access memory takes the form of integrated circuits. RAM is normally associated with volatile types of memory where stored information is lost if power is removed,

If RAM fills up, the processor needs to continually go to the hard disk to overlay old data in RAM with new, slowing the computer's operation. Unlike a hard disk, which can become completely full of data and unable to accept any more, RAM never runs out of memory, but the combination of RAM and storage memory can be completely used up.

DRAM vs. SRAM:

RAM comes in two primary forms:

Dynamic random access memory. DRAM is what makes up the typical computing device RAM and, as noted above, requires constant power to hold on to stored data.

Static random access memory. SRAM doesn't need constant power to hold on to data, but the way the memory chips are made means they are much larger and thousands of times more expensive than an equivalent amount of DRAM. However, SRAM is significantly faster than DRAM. The price and speed differences mean SRAM is mainly used in small amounts as cache memory inside a device's processor.

LECTURE 29:

ROM:

Read-only memory (ROM) is a type of non-volatile memory used in computers and other electronic devices. Data stored in ROM can only be modified slowly, with difficulty, or not at all, so it is mainly used to store firmware (software that is closely tied to specific hardware and unlikely to need frequent updates).

More recently, *ROM* has come to include memory that is read-only in normal operation, but can still be reprogrammed in some way. Erasable programmable read-only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM) can be erased and re-programmed, but usually this can only be done at relatively slow speeds, may require special equipment to achieve, and is typically only possible a certain number of times

So, there are various types of ROM-

MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs, which are inexpensive.

PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM program. Inside the PROM chip, there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

EPROM (Erasable and Programmable Read Only Memory)

EPROM can be erased by exposing it to ultra-violet light for a duration of up to 40 minutes. Usually, an EPROM eraser achieves this function. During programming, an electrical charge is trapped in an insulated gate region. The charge is retained for more than 10 years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use, the quartz lid is sealed with a sticker.

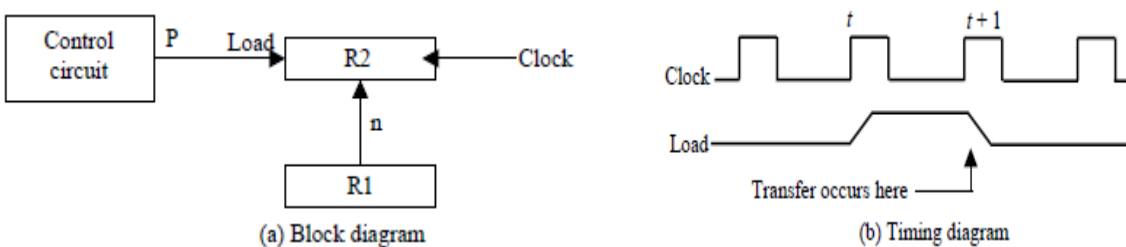
EEPROM (Electrically Erasable and Programmable Read Only Memory)

EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of reprogramming is flexible but slow.

LECTURE 30

REGISTER TRANSFER

Copying the contents of one register to another is a **register transfer**. Register Transfer Language is the symbolic notation used to describe this micro operation transfer. A register transfer is indicated as $R2 \leftarrow R1$: In this case the contents of register R1 are copied (loaded) into register R2. A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse. Often this transfer occurs under a control condition. If the signal is 1,

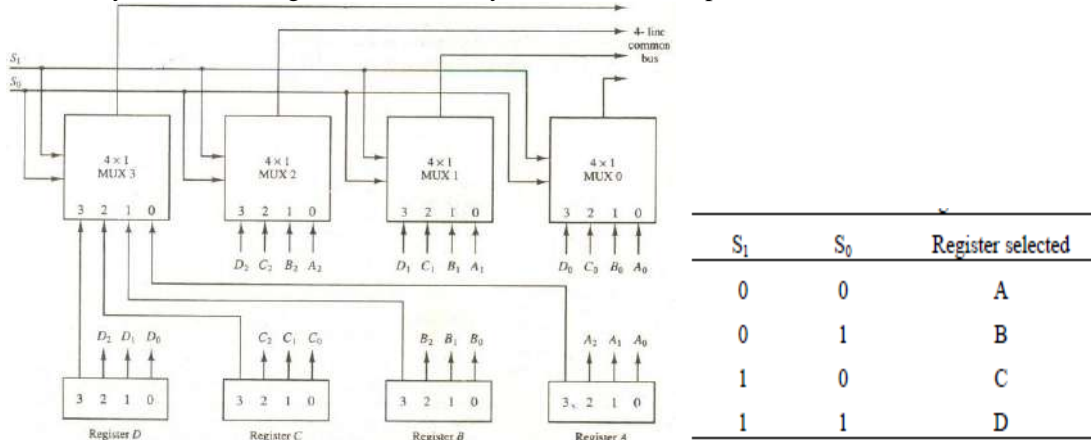


the action takes place. This is represented as $P: R2 \leftarrow R1$ which means if $P = 1$, then load the contents of register R1 into register R2. The same clock controls the circuits that generate the control function and the destination register. Registers are assumed to use positive-edge-triggered flip-flops.

BUS AND MEMORY TRANSFERS :

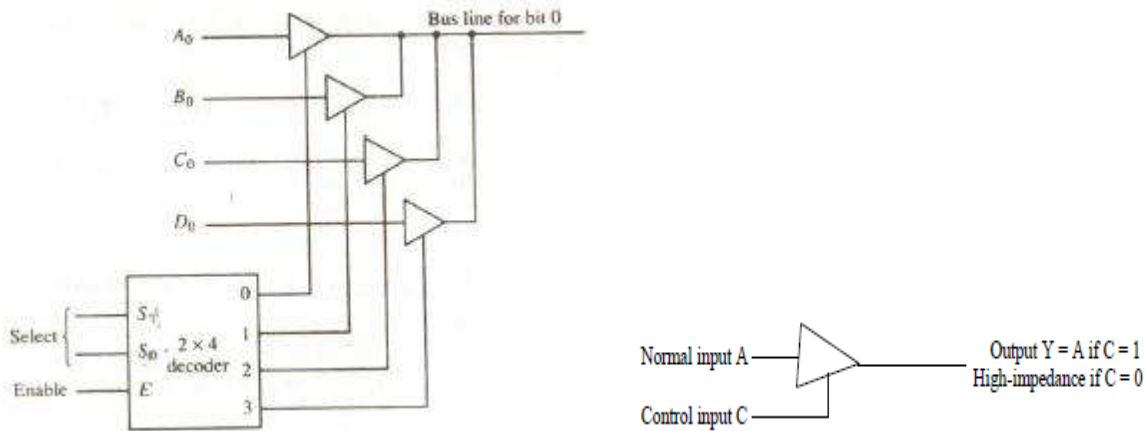
A digital computer has many registers, and paths must be provided to transfer in form one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme is a common bus system. A bus structure 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 during each particular register transfer.

One way of constructing a common bus system is with multiplexers.



THREE-STATE BUS BUFFERS

A bus system can be constructed with three-state gates instead of multiplexes. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance state behaves like an open circuit.



MEMORY TRANSFER

The transfer of information from a memory word to the outside environment is called a read

operation. The transfer of new information to be stored into the memory is called a write operation. The address register (AR) is used to select a memory address, and the data register (DR) is used to send and receive data.

Read: $DR \leftarrow M[AR]$: This causes a transfer to information into DR from the memory word M selected by the address in AR.

Write: $M[AR] \leftarrow R1$: This causes transfer of information from a register R1 into the memory word M selected by the address in AR.

LECTURE 31:

Control Unit:

The **control unit** (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices on how to respond to a program's instructions.

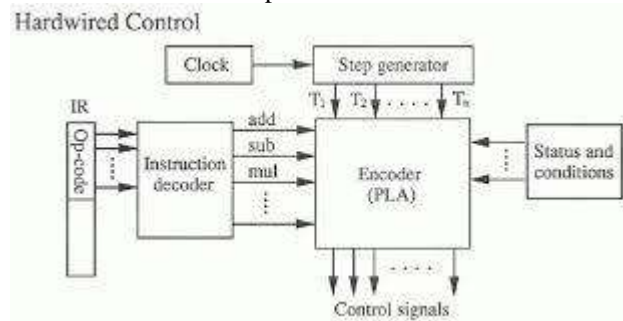
There are two design approaches for CU: Hardwired approach and Micro-programming approach

Hardwired Control Unit:

The control signals are generated by the help of the hardware. It can be designed as the clock sequential circuit. It is implemented with logic gates, flip-flops, decoders, multiplexers and other logic buildings blocks. Hardwired control units are generally faster than microprogrammed designs.

Their design uses a fixed architecture—it requires changes in the wiring if the instruction set is modified or changed. This architecture is preferred in reduced instruction set computers (RISC) as they use a simpler instruction set.

A controller that uses this approach can operate at high speed; however, it has little flexibility, and the complexity of the instruction set it can implement is limited.

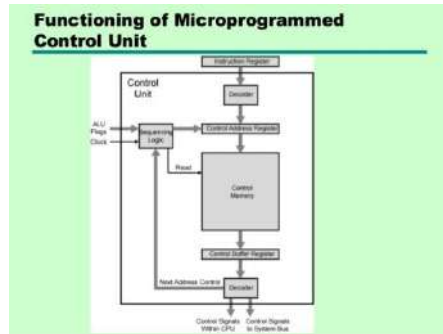


The hardwired approach has become less popular as computers have evolved. Previously, control units for CPUs used ad-hoc logic, and they were difficult to design.

Micro programmed Control Unit:

In micro programmed control unit, the logic of the control unit is specified by a micro program. A microprogram consists of a sequence of microinstructions. Microinstructions are stored in control memory.

A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.



All controls that can be activated simultaneously are grouped together to form the control words. The individual control words in this microprogram are referred to as microinstructions.

Comparison

Attributes	Hardwired Control	Microprogramming Control
Speed	Fast	Slow
Cost of Implementation	More	Cheaper
Flexibility	Difficult to modify	Flexible
Ability to handle complex instruction	Difficult	Easier
Decoding	Complex	Easy
Application	RISC	CISC
Instruction Set Size	Small	Large
Control Memory	Absent	Present

Lecture 32

Input-Output (I/O) is the means of interaction between the computer system and outside world. Input-output devices are the external devices connected with the computer system (CPU, Memory). There are various kinds of input devices such as keyboard, optical input devices (Card Reader, Paper Tape Reader etc.), magnetic input devices (magnetic stripe reader etc.), and screen input devices (Touch Screen, Light Pen, Mouse etc.). On the other hand, CRT, card puncher, paper tape puncher, printer, plotter are various kinds of output devices. Every I/O device has its own interface to communicate with the outside world via system bus. System bus consists of address, data and control bus. Below figure (fig.1) shows the connection between I/O devices and system bus.

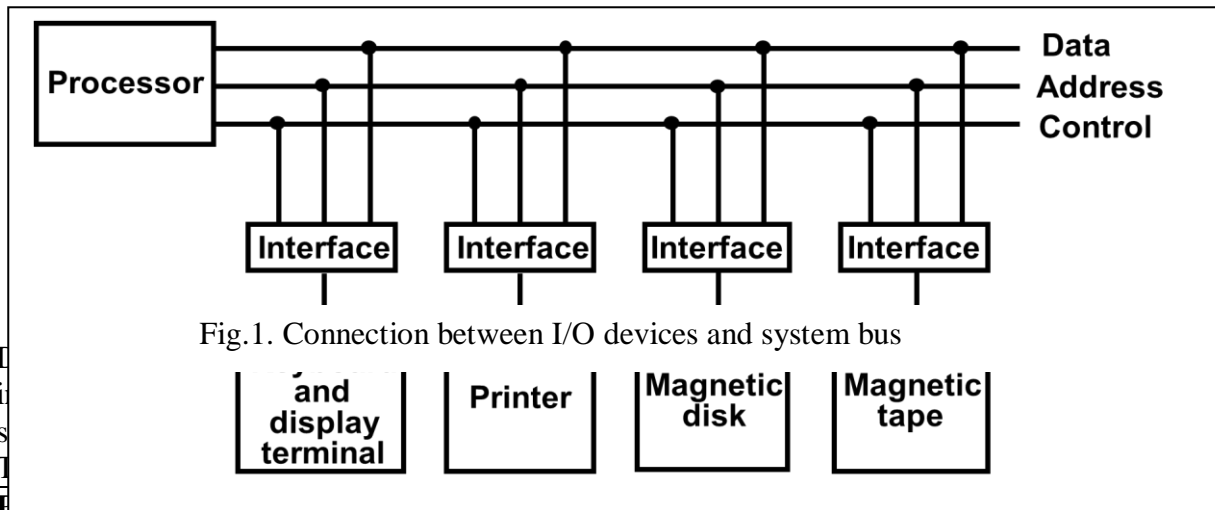
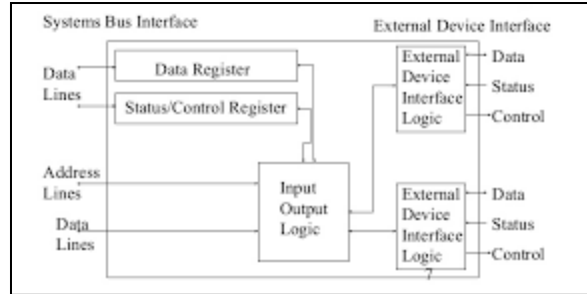


Fig.1. Connection between I/O devices and system bus

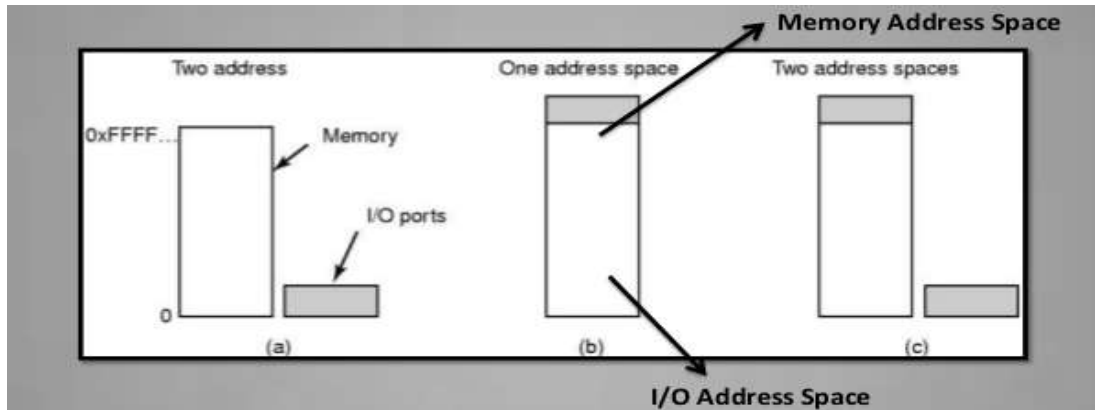
	Electromechanical Devices	Electronic Device
Device type	Electromechanical Devices	Electronic Device
Data Transfer Rate	Usually slower	Usually faster than peripherals
Unit of Information	Byte	Word
Operating Modes	Autonomous, Asynchronous	Synchronous

The processor places a particular address on the address bus for either input or output device. The interface of each peripheral device decodes the address. A particular interface, whose address is mentioned on the address bus, activates the path between system bus and peripheral device. The processor places commands (operation) on the control bus and data is placed on the data bus. Corresponding peripheral interface decodes the commands, provides signals for the peripheral controller and synchronizes the data flow.

I/O Mapping

There are three methods of interaction between peripheral device and CPU. They are

- a. Separate address, data and control bus
- b. Common address, data and control bus (Memory mapped I/O)
- c. Common address and data and but separate control bus(Isolated I/O)



Memory mapped I/O

- Common address, data and control bus.
- Devices (I/O) and memory share an address space from main memory.
- Instructions (move, add, sub, load, store etc.) which are used for memory can be used for I/O operation. No distinct commands for I/O.
- Large number of commands set.
- Faster operation.

Isolated I/O

- Common address and data bus but separate control bus.
- Common address space is divided into two separate spaces. One space is for memory and other one is for I/O.
- Both memory and I/O share the buses on time-sharing mode.
- Distinct commands for I/O and memory
- Limited commands set.
- Slower operation

Lecture 33

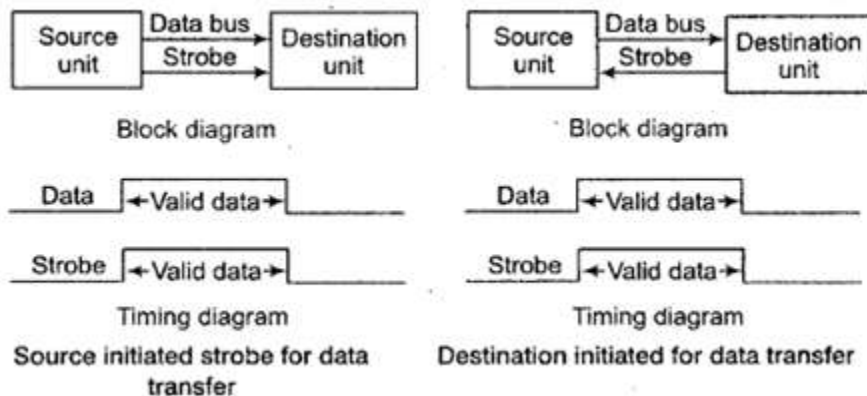
Synchronous and Asynchronous Data Transfer

The transfer between processor and peripheral devices generally occurs in the presence of common clock which resides in the interface register. Sender and receiver can work simultaneously in presence of global/shared clock. This approach is known as synchronous transfer. In the absence of clock, control signals are used for initiating the communication between processor and peripheral devices by indicating starting time of transmission. This method is termed as asynchronous transfer. Asynchronous transfer has two types of control signals known as strobe control and handshaking.

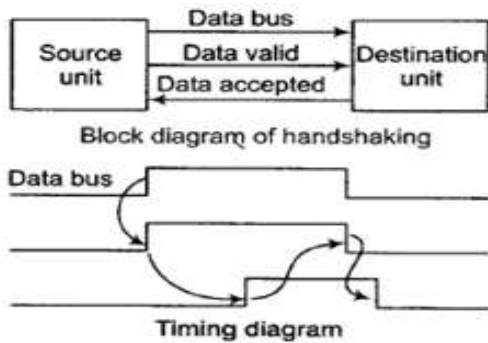
Strobe control is a control/pulse signal initiated by either by source or destination unit. After initiating strobe signal, source unit places the valid data on the data bus for destination. On the other hand, destination unit initiates the strobe pulse to inform the source to place the valid data on the data bus.

But in strobe control approach, source is unaware of the fact that whether destination has actually received the data at all. To overcome the situation, two control signals are used in handshaking approach. They are data valid and data accepted. Data valid signal informs the destination that sender has placed the valid data on the data bus. On the other hand, destination unit sends data accepted signal to inform the sender that destination has received the data successfully.

Strobe Control



Handshaking



Lecture 34

Modes of Transfer

The information is shared between peripheral devices to processor by means of memory unit. There are three modes of transferring data between processor and peripheral devices. They are

1. Programmed I/O
2. Interrupt- initiated I/O
3. Direct Memory Access (DMA)

Programmed I/O

CPU issues read commands and stays in the whole process until data transfer is complete. As CPU is faster than I/O module and I/O operation is performed by slower I/O module, it has to wait for completion of the I/O operation.

Programmed I/O works in these ways:



- CPU issues request for I/O operation
- I/O module checks the status of the peripheral device by looking at the status bits
- CPU checks status bits for availability of the peripheral device
- Based on availability of the device, I/O module places the data on the data bus
- CPU reads the data from I/O module and writes into memory
- I/O module neither notify nor interrupt CPU directly
- CPU may wait or come back later in case I/O module is unavailable
- During waiting phase, CPU checks status bits repeatedly

But it is inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory. Moreover, CPU has to wait for long time as I/O module is slower and on availability of peripheral device, I/O module can start I/O operation.

Interrupt-initiated I/O:

CPU initiates the data transfer process and does not need to wait for I/O module to complete the task unlike programmed I/O. In this approach CPU can proceed to its normal work after issuing read command until interrupted by I/O module.

Interrupt-initiated I/O works in these ways:

- CPU issues read command to I/O module
- I/O module checks the status of peripheral device while CPU proceeds for other work
- I/O module interrupts CPU, CPU temporarily stops its other task and resume its current work
- CPU requests for data from I/O module
- I/O module transfers data and CPU writes data into memory



Although Interrupt relieves the CPU, CPU does not need to wait for long to get the service from peripheral device. Again, it is still inefficient in data transfer of large amount because the CPU has to transfer the data word by word between I/O module and memory.

When CPU is able to know the address of the Interrupt Service Routine (ISR) to carry out in software (memory) in advance and interrupting device directs the processor to the appropriate interrupt service routine, this is known as vectored interrupt. For example,

when 8085 is interrupted with RST 5.5 pin, it automatically directs PC to the address 002CH.

A non-vectored interrupt is where CPU is not able to know address of the Interrupt Service Routine (ISR) and the interrupting device never sends an interrupt vector. So, interrupt is received by the CPU, and it jumps the program counter to a fixed address in hardware. For example, CALL instruction in 8085.

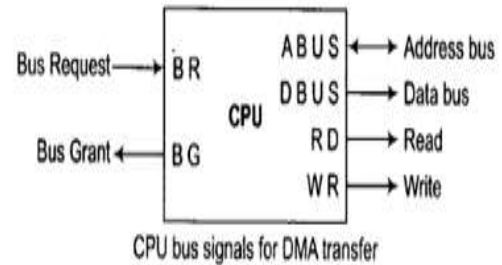
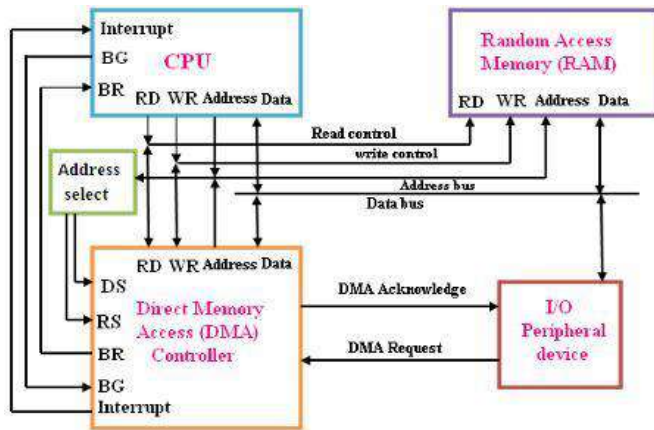
DMA (Direct Memory Access)

In this approach data transfer can be done between memory and peripheral device without intervention of CPU. CPU initiates the data transfer and afterwards it gives the authority to Direct Memory Access (DMA). DMA starts transferring of data between memory and peripheral device directly. At the end of exchanging the entire data block, CPU is interrupted by DMA. So CPU is involved at the beginning and end of the transfer. For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. 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. For example, a PCI controller and a hard drive controller each have their own set of DMA channels.

DMA is efficient for transferring large amount of data between memory and peripheral device.

DMA works in these ways

- The CPU places address on the address bus and the number of words (bytes) needed to be transferred, on the data bus. Thus CPU initiates the transfer and proceeds with other task.
- After the transfer is made, the DMA requests to the memory controller for memory cycles through the memory bus. This request is granted by the memory controller and the DMA can start transferring the data directly into memory.
- DMA requests CPU by issuing Bus Request (BR) to get the control of buses.
- When this BR is active, the CPU stops the execution of the current instruction and places the address bus and the data bus. The CPU activates the Bus Grant (BG) output to inform the external DMA that the buses are available.
- The DMA now gets the control of the buses to start the memory transfer. When DMA terminates the transfer, it disables the bus request line.
- The CPU disables the bus grant, takes the control of the buses.



Hardware design is complicated because the DMA controller must be integrated into the system, and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

Types of Data transfers: The DMA Controller has several options available for the transfer of data. They are:

- Burst transfer
 - Block sequence consisting of memory words is transferred in a continuous bus when DMA controller is the master
- 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

Lecture 35

Bus Arbitration

Bus arbitration means assigning the authority to use the bus to only one processor among multiple processors. In a situation where multiple processors and multiple single purpose processor/controller are present to share the bus, bus can be granted to only one processor. Any one controller or the processor can be the bus master at a time. Current bus master accesses the bus and leaves the control of the bus to next bus requesting processor.

There are three types of bus arbitration methods. They are

- Daisy Chain method
- Independent Bus Requests and Grant method
- Polling method

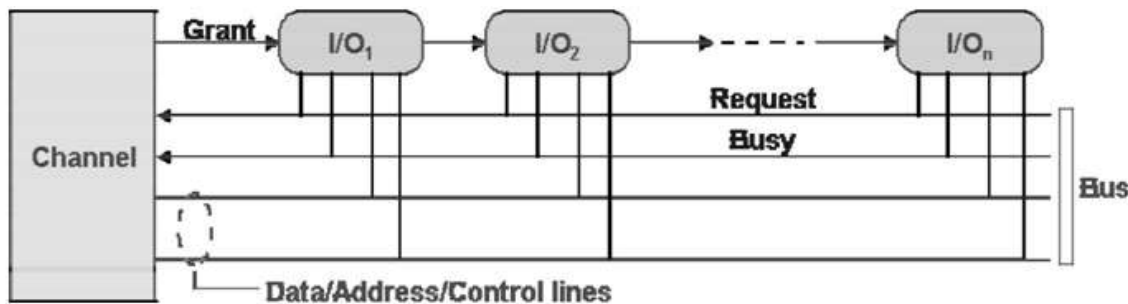
Daisy Chaining

Daisy chaining has two approaches. They are

- Centralized control (priority scheme)
- Decentralized control (Round-robin Scheme)

Centralized control (priority scheme)

The I/O devices send the request signal to get the access of the bus. The channel activates and sends a Grant signal to the first I/O device (closest to the channel) provided any device does not access the bus and there is no signal on busy line. The first device propagates the Grant signal to the next I/O device if the first device has not made any request to access the bus.

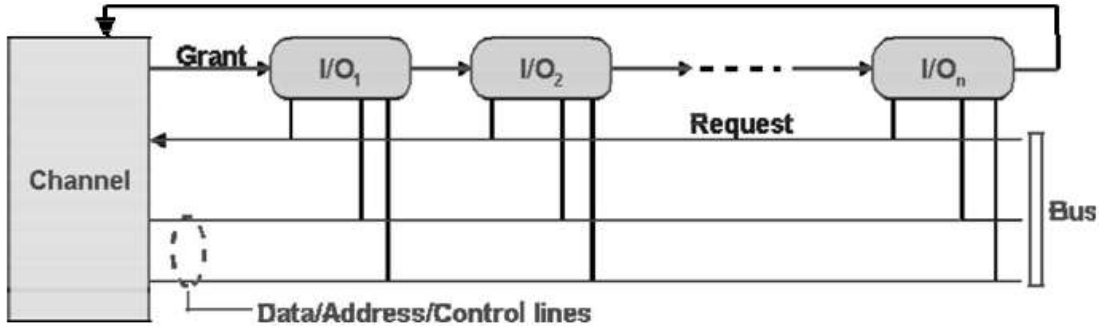


This propagation continues to the rest of the devices until the requested I/O device gets the bus access. After the task completed by the requesting I/O device, it resets the busy line to inform others that any device can now access the bus. The device having grant signal, can access the bus. In this method, highest priority device is nearest to the channel and lowest priority device is placed in the far away from the channel.

Decentralized control (Round-robin Scheme)

The I/O devices send the request signal to get the access of the bus. The channel activates and sends a Grant signal to the first I/O device (closest to the channel) provided any device does not

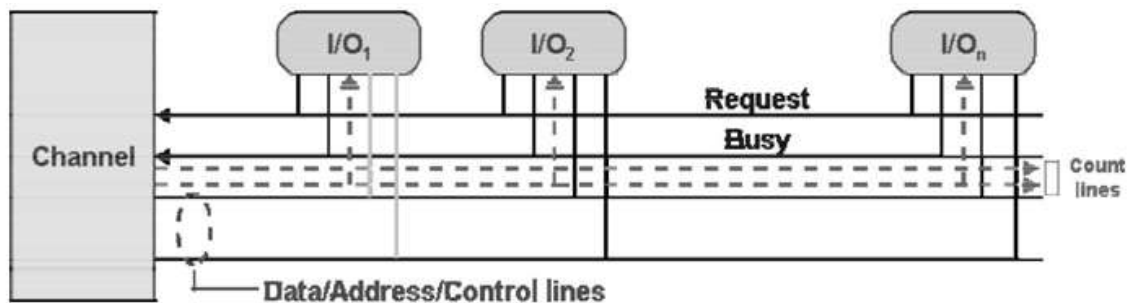
access the bus and there is no signal on busy line. The first device propagates the Grant signal to the next I/O device if the first device has not made any request to access the bus.



This propagation continues to the rest of the devices until the requested I/O device gets the bus access. After the task completed by the requesting I/O device, it checks if the request line is activated or not. If request line is activated, the current device sends the Grant signal to the next I/O device in round-robin mode and the process continues. Otherwise, the current device resets the busy line to inform others that any device can now access the bus.

Polling

The device which wants to get access the bus, places the bus request signal on the bus. The channel starts interrogating the devices one by one to check for bus request signal if the busy signal is off. It does this by sequentially sending a count from 1 to n on $\log_2 n$ lines to the devices.

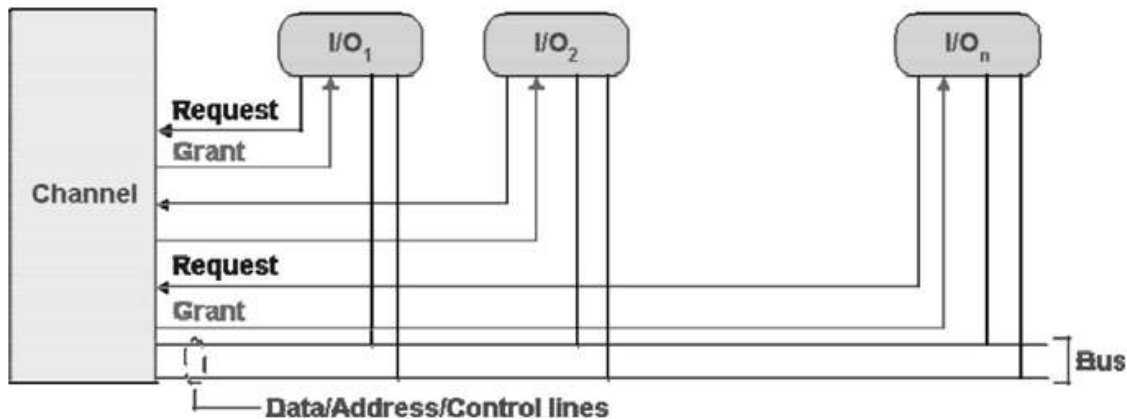


Whenever a requesting device compares the count and if count is matched against its own number (address), it activates the busy line. The channel then stops the count

and the device has got access on the bus. When access is completed by the requesting device, the busy line is disabled and the channel can start counting either from the last device (Round-Robin) or start from the beginning (priority).

Independent Requests

There is a separate request and grant signal for each of the I/O device. The device which wants to get access the bus, places the bus request signal on the bus.



The channel responds by granting access to the requesting device. The device having grant signal, can access the bus. When a device finishes access, it disables request signal. The channel can use either a priority scheme or round-robin scheme to grant the access.

Lecture 36

Input Output Processor (IOP)

- The I/O processor can execute specialised I/O program residing in the memory without involvement of the CPU.
- Processor having DMA capability and interrupt facility, can communicates with I/O devices
- IOP takes care of input and output tasks and CPU does not need to involve in I/O operation.
- CPU is the master and IOP is the slave.
- IOP can fetch and execute its own instructions
- An advanced I/O processor can have its own memory, it can control a large set of I/O devices without much involvement from the CPU.
- In computer systems having IOPs, the CPU generally does not execute I/O data transfer instructions.
- I/O instructions are stored in memory and are executed by IOPs.

