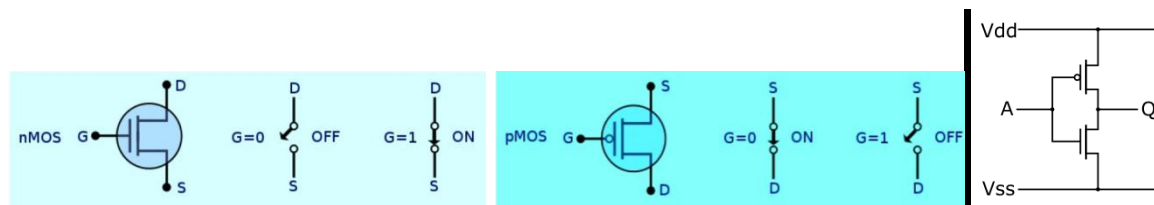


Unit-1

**CMOS Technology:** Introduction to MOS transistors and VLSI fabrication(NMOS, PMOS, CMOS and BiCMOS)- Introduction to power reduction techniques-Dynamic Power Reduction-Static Power Reduction- NMOS and CMOS inverter- Determination of pull up to pull down ratios – propagation delays – power dissipation - Stick Diagram -MOS layers - design rules and layout- choice of layers and Scaling.

**Basic Introduction:**

**CMOS** or Complementary Metal Oxide Semiconductor is a combination of NMOS and PMOS transistors. NMOS is an N-type Metal Oxide Semiconductor, and PMOS is a P-type Metal Oxide Semiconductor. N-type is a type of pentavalent impurities, and P-type is a type of trivalent impurities doped on the semiconductor. The three terminals of the transistors are Gate (G), Source (S), and Drain (D). The doping of p-type/n-type is applied on the D and S terminals.



Basis Difference	of CMOS Technology	NMOS Technology
Full form	CMOS stands for Complementary Metal Oxide Semiconductor.	NMOS stands for N-channel Metal Oxide Semiconductor.
Definition	A metal oxide semiconductor technology that combines both PMOS and NMOS technologies is called CMOS.	A metal oxide semiconductor technology that uses N-type channel between source and drain terminals is called NMOS.
Operation	The CMOS performs its operation by employing symmetrical as well as complementary pairs of P-type and N-type MOSFETs.	The NMOS performs its operation by making an inversion layer within a Ptype substrate.
Logic level	The logic level of CMOS is 0 V / 5 V.	The logic level of NMOS depends on the $\beta$ ratio as well as noise margins.
Layout	CMOS has more regular layout.	NMOS has irregular layout.
Power dissipation	In case of CMOS, the power dissipation is zero, when it is in standby mode.	The power dissipates in NMOS, when its output is zero (0).
Power supply	For CMOS, the power supply may vary from 1.5 V to 15 V.	For NMOS, the power supply is fixed depending on $V_{DD}$ .
Packing density	CMOS has less packing density. Where, it requires 2N devices for N inputs.	The packing density of NMOS is high. It requires (N+1) devices for N inputs.
Load to drive ratio	CMOS has load / drive ratio 1:1 or 2:1.	NMOS has load / drive ratio 4:1.
Transmission	The transmission gate of CMOS allows to	The transmission gate of NMOS allows to

gate	pass both '0' and '1' logic well.	pass only the logic '0' well. If it pass logic '1', then it will have VT drop.
Static power consumption	CMOS consumes low static power.	NMOS consumes relatively more static power.
Noise immunity	CMOS has high noise immunity.	NMOS has comparatively low noise immunity.
Applications	The CMOS is used to design various types of digital logic circuits, microprocessors, microcontrollers, memories, etc.	NMOS is used to design several types of digital logic circuits such as microprocessors, memory chips, and many other MOS devices.

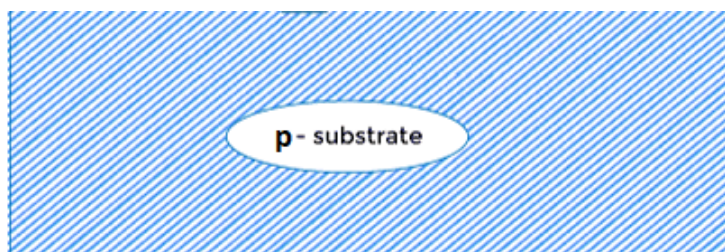
## 1. NMOS Fabrication

The doping on the NMOS will be a Pentavalent (five valence electrons) impurity, such as **boron** and **antimony**. It has a p-substrate on which the n-type channel is created. As the name implies, the majority carriers participating in the current are electrons. The movement of electrons is fast as compared to the holes. Thus, NMOS is faster than the PMOS.

The NMOS fabrication includes eight steps, which are listed as follows:

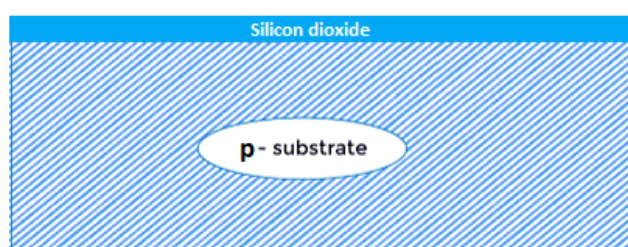
### Step 1: Processing the substrate

The first step is to create a **p-type substrate**. The p-type has trivalent impurities (three valence electrons), such as boron with a concentration upto  $10^{16}/\text{cm}^3$ . A pure thin film of the silicon wafer is selected on which the p-type impurities are applied as crystals. The diameter of the wafer can be upto 0.15m or 150mm. The chosen wafer material is silicon because it is a clean and high-quality semiconductor material preferred for fabrication.



### Step 2: Silicon dioxide layer

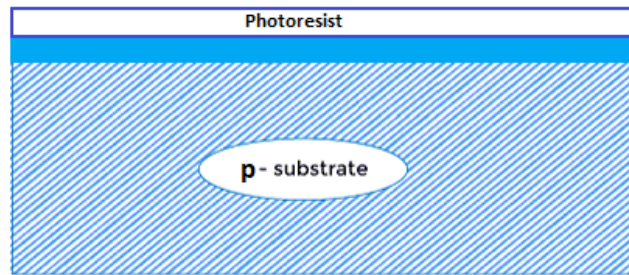
Silicon dioxide is made up of two materials silicon (Si) and oxygen ( $\text{O}_2$ ). It is also known as **oxide**. Silicon has a stable structure and is considered as the most abundant metal available on the Earth. It is combined with oxygen that acts as an insulator or conductor under various conditions.



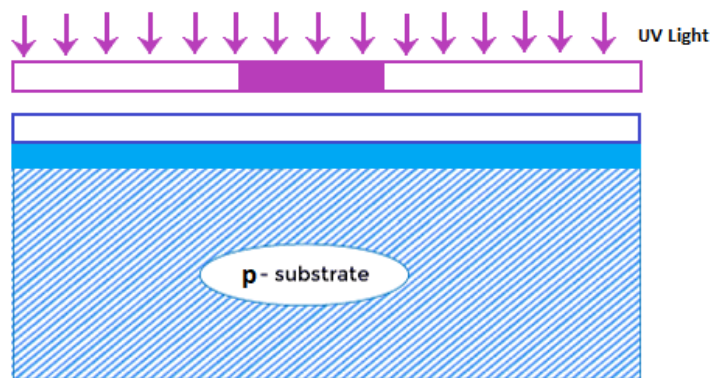
$\text{SiO}_2$  layer is grown over the surface of the p-type substrate to prevent it from external factors. It also acts as a barrier to the dopants applied on the layer during the processing. The silicon dioxide thickness is very small, around  $0.000001\text{m}$  or  $1\mu\text{m}$ .

**Step 3:** Photoresist material is applied on the SiO<sub>2</sub> layer

The silicon dioxide layer is covered with the photoresist material. It is a **light-sensitive** material that forms the coating over the surface of the SiO<sub>2</sub> layer. It is useful in reducing the size of the transistors.

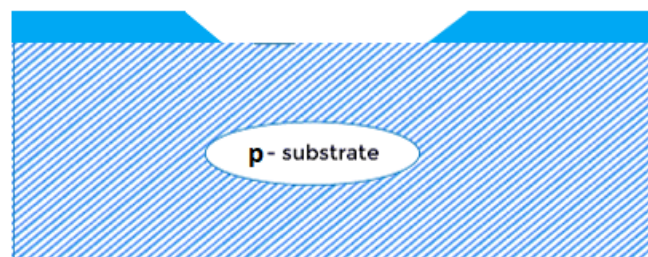


After the photoresist is applied on the silicon dioxide layer, a mask with the desired pattern is used as a medium to expose UV (Ultra-Violet) lights. The UV light through the mask reaches the photoresist material. The exposed resist remains on the surface and the unexposed part is removed from the surface.



**Step 4:** Etching the regions

The unexposed window is removed from the surface and the regions are etched together to form a clean wafer surface. It is shown below:



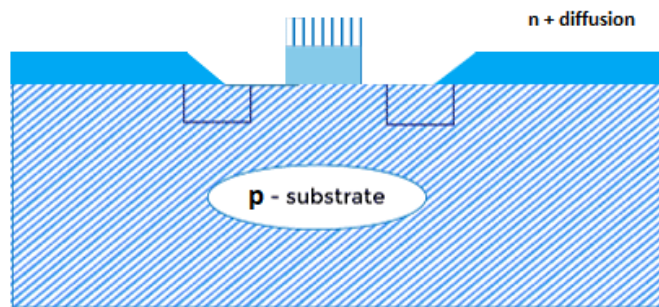
**Step 5:** Formation of Gate

The remaining photoresist layer is removed from the wafer. A thin silicon dioxide layer of 0.0000001m or 0.1 um is grown over the surface. The polysilicon is further added to the surface that forms a gate structure, which is deposited by CVD process. The Chemical Vapor Deposition produces solid materials of high quality. The polysilicon is preferred for the gate because of its high melting point. Its properties are also similar to that of SiO<sub>2</sub>.

Note: The number of doping concentrations, the thickness of the layer, and the resistivity are the three essential elements to be considered for the fabrication process.

**Step 6:** Creating the area for drain and source terminals

The thin oxide layer on the surface of the silicon wafer is removed and the n-type impurities are inserted with the help of the diffusion process in the specified exposed area. It forms the n-channel at the source and drain terminals. The wafer is first heated at a very high temperature and the gas is passed into it. The area exposed is filled with the gas containing n-type impurities, such as phosphorous.

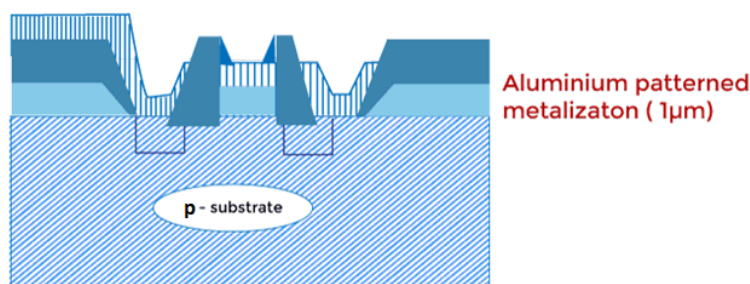


**Step 7:** SiO<sub>2</sub> and photoresist is again deposited on the source and drain terminals

The same process is again carried to protect the S and D terminals. The oxide layer was removed from the surface of the wafer to create the two terminals, as discussed in step 6. The silicon dioxide and the photoresist are deposited, etched, and masked to protect it. The contact holes are left exposed for the connections.

**Step 8:** Making of the metal layer

It is almost the last step of the NMOS fabrication process. The metal layer of aluminum is deposited on the surface of wafer including the contact holes. The thickness of the aluminum is around 1µm. The metal layer is further masked and etched to form the required interconnection pattern.



nMOS fabrication process

The layer of different materials was applied to the silicon wafer at each step. Thus, the NMOS fabrication process involves the deposition of four major layers. It includes silicon dioxide, photoresist, polysilicon, and the aluminum metal layer.

\*\*\*\*\*

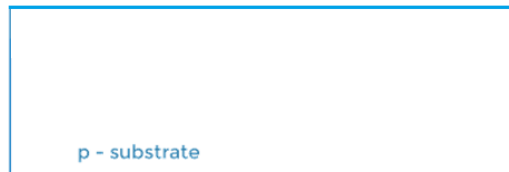
## 2. N-well Fabrication Or CMOS Fabrication process.

It is a CMOS fabrication process. It means that the PMOS and NMOS are fabricated in different ways. PMOS is created by placing it in the n-well that has a p-type channel. The NMOS is created similarly as discussed above, i.e., on the substrate. Hence, the fabrication of CMOS is known as N-TUB.

The steps involved in the N-TUB fabrication process are as follows:

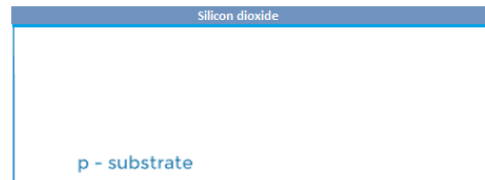
### 1. Wafer

The silicon wafer is selected for doping p-type impurities on it. The doped wafer formed will be the p-type substrate with the trivalent impurities.



## 2. Oxidation of wafer

The Silicon dioxide layer or oxide layer is created on the surface that protects the substrate. Silicon is one of the metals that are easily available with properties suitable for the fabrication process.



## 3. Photoresist deposition

The photoresist material is deposited on the wafer. It allows the formation of the n-well on the p-type substrate.

## 4. N-well Mask

The n-well mask is exposed on the wafer with a particular pattern. The soft part or the unexposed part of the photo resist material is removed to expose the SiO<sub>2</sub> layer.

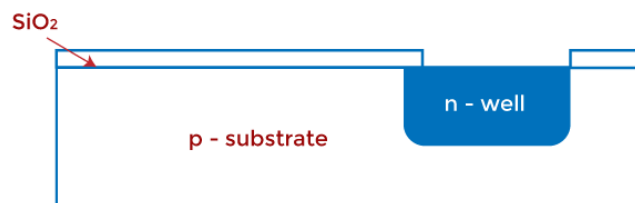
## 5. Oxide Etch

As discussed, the right part of the wafer is not covered by the photoresist. It is due to the area left for the formation of the n-well. To protect it, the oxide layer is etched with the **Hydrofluoric acid**. The silicon atoms of the SiO<sub>2</sub> layer and the fluoride atoms of HF acid form the strong bond. The silicon wafer is now ready to be exposed to the n-well area.

The leftover photoresist material is removed with the help of the etching process.

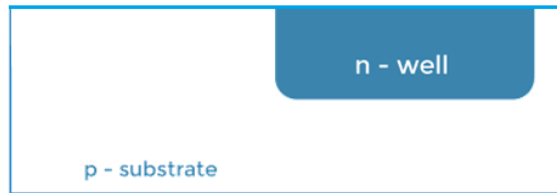
## 6. N-well formation

The diffusion process is used to make the n-well on the right side of the wafer. Diffusion is a method of adding impurities from a high concentration region towards a low concentration region. We can also use the ion implantation process to create an n-well.



## 7. Oxide removal

The remaining oxide on the surface of the wafer is stripped off with the help of HF acid. The oxide protects and needs to be removed to create a gate junction. It is done to expose the n-well formed in the above step directly.



### 8. Gate formation

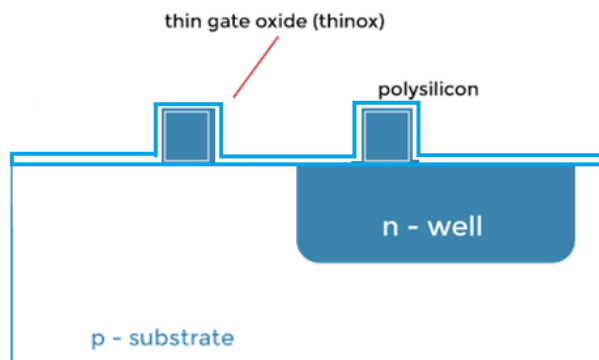
The polysilicon is added to the surface that forms a gate structure deposited by CVD process. It is a heavily doped layer of polysilicon present over the thinox, which is a type of thin gate oxide.

### 9. Poly patterning

The wafer surface is exposed with the photoresist and the mask to create the two G terminals. It is because CMOS is a combination of NMOS and PMOS, and both have separate gates, as shown below:

### 10. Diffusion pattern

A protective layer of oxide and photoresist is again applied on the wafer to protect the two gate terminals.



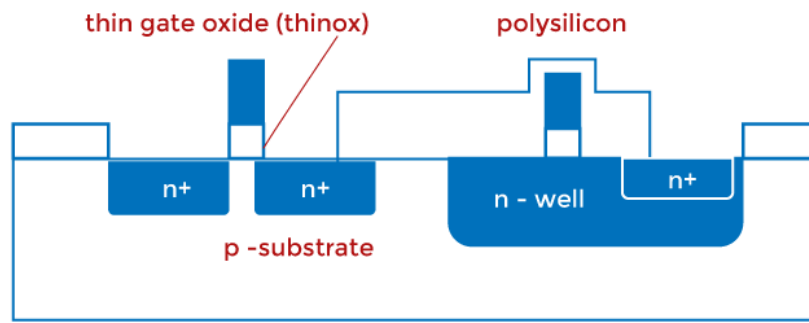
### 11. Creating area for source and drain

To create the area for S and D, the protective oxide is removed present on the surface of the wafer. It creates the two vacant areas.

### 12. N-diffusion regions

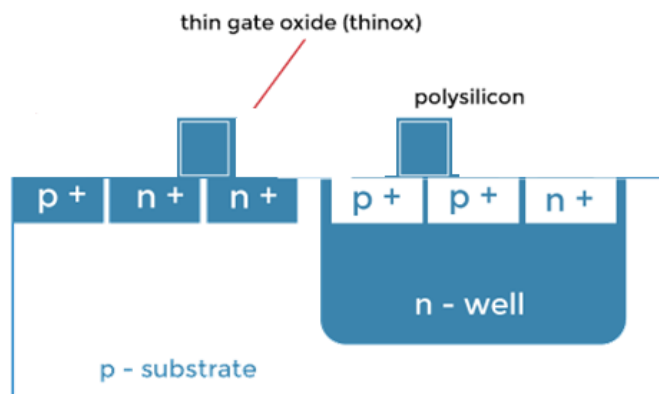
The two vacant areas include two source terminals and two drain terminals on each side. The n+ diffusion regions are created by injecting the n+ impurities to those vacant areas. It automatically forms the S and D terminals adjacent to the gate. The n-well on the right side is also created through n+ diffusion.

The wafer is heated at a high temperature to create the n-well regions through the diffusion process.



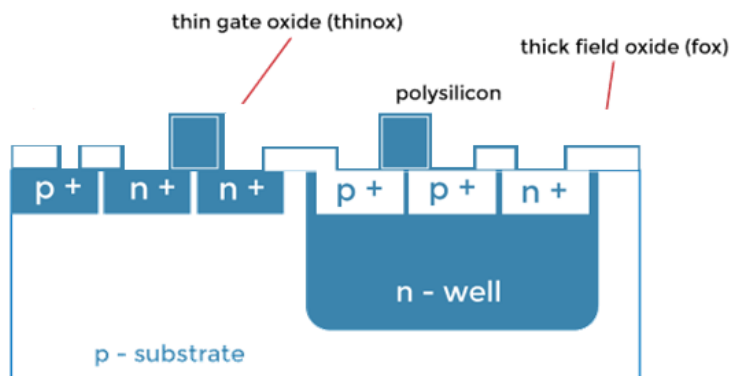
### 13. P-diffusion

The NMOS has n+ S and D regions, while the PMOS has p+ regions. The n+ regions are already created with the help of above step. Next, the p+ diffusion mask is used that completes the formation of all the active regions of the MOS transistors.



### 14. Field oxide

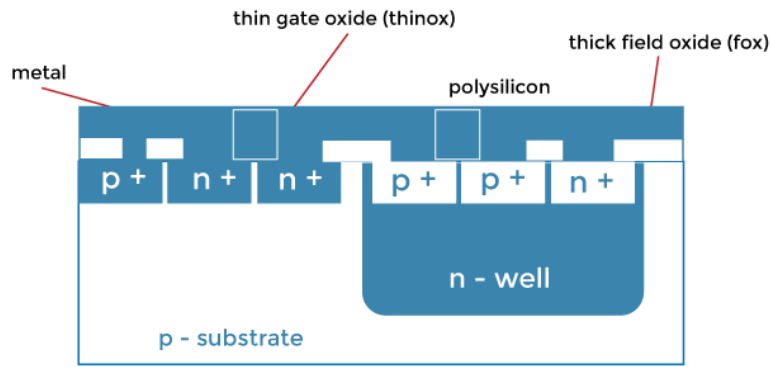
To insulate the wafer, the field oxide ( $\text{SiO}_2$ ) is deposited on its surface.



### 15. Metal formation

The metal layer of aluminum is deposited on the surface of wafer including the contact holes. It also fills the cut holes. It is patterned with the help of metal mask.

The entire process of the N-TUB fabrication can be summarized with the help of flow diagram shown below:

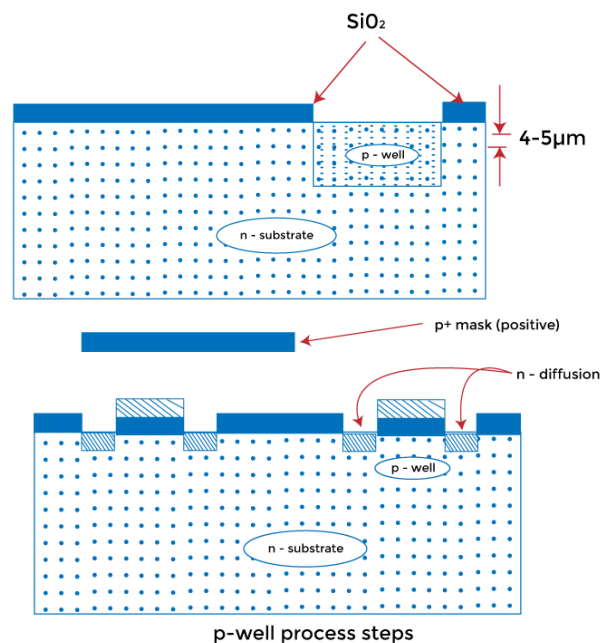


The above process shows the formation of NMOS (left side) and the PMOS transistors. We can also create an inverted circuit fabricated using the same steps. It consists of the PMOS and the NMOS (right side) transistors. The NMOS transistors perform better as compared to PMOS. Thus, n-wells are created to overcome the poor performance of the PMOS.

### 3. P-TUB Fabrication Or P-well Fabrication

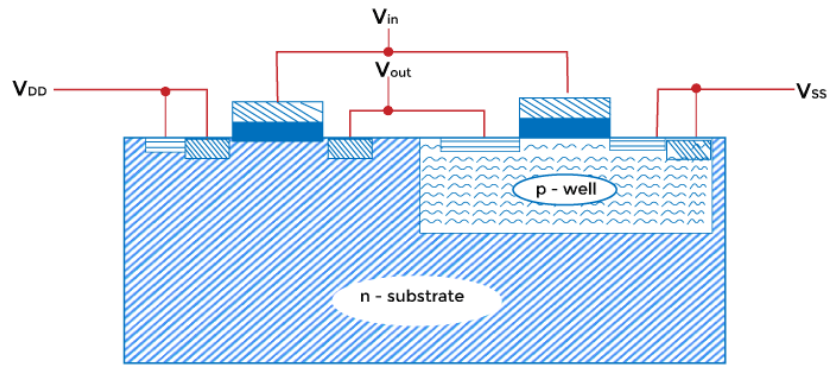
It is also a CMOS fabrication process. The NMOS transistors are created by placing them in the **p-well** with a p-type channel. The PMOS is created on the n-type substrate. Hence, the fabrication of NMOS is known as P-TUB.

Here, the substrate is n-type, and the doping on the source and drain regions is p-type. A p-well is diffused into the substrate through the diffusion process. The doping concentration and depth of the p-well affect the n-type devices' voltage. Thus, special care is required. The deeper the wells, the larger the surface area it may require.



P-TUB combines the PMOS and the NMOS (right side). The n-type acts as a substrate for the PMOS, and the p-well acts as the substrate for the NMOS. These two areas are electrically isolated. The p-well CMOS inverter is shown below:

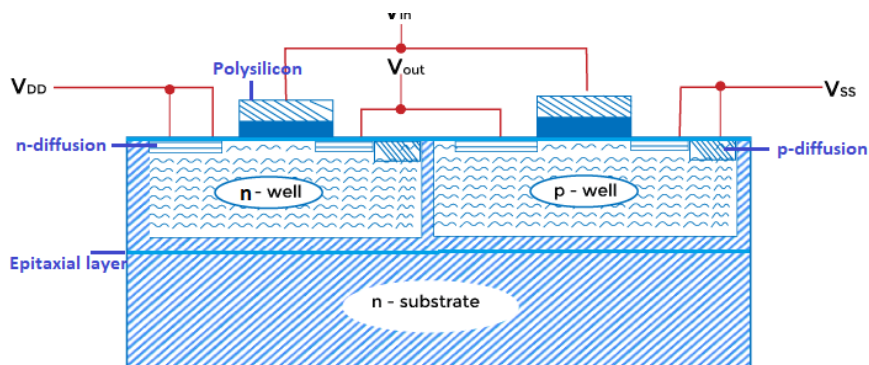




The masking, diffusion, etc. are similar to that of N-TUB. The various mask used in the steps for fabrication are used to defines the areas for deep wells, thick stripped oxide deposition, contacts areas, interconnections, deposition of polysilicon layer, p-diffusions, and metal layer patterns.

#### 4. Twin-TUB Fabrication

Twin-TUB is a combination of p-well and n-well processes formed on the same substrate. It is also known as **dual well process**. The high resistivity n-type substrate with both n-well and p-well regions is created, as shown below:



It is an inverting arrangement of twin-tub. The separate transistors and their arrangement help to optimize the n-type devices, p-type devices, and other parameters, such as body effect and threshold voltage. The wafer has two layers. The top layer is the epitaxial layer and the main substrate layer is of n-type.

The steps of the twin-TUB fabrication are as follows:

**Step 1:** Deposition of thin oxide layer of silicon dioxide ( $\text{SiO}_2$ ).

**Step 2:** The deposition of **silicon nitride** layer using the CVD process. It has various advantages, such as high temperature stability and light weight.

**Step 3:** The third step includes creating trenches and filling them with  $\text{SiO}_2$ , which is an insulating material. The trenches are created to prevent the current leakage.

**Step 4:** The oxide and nitride is removed to deposit the n-well and p-well regions with the help of diffusion.

**Step 5:** The p-well and the n-well mask are used to define the specific areas on both sides. The implant and annealing are required to adjust the doping concentration of the two wells. The annealing process reduces the hardness for efficient doping.

**Step 6:** A thin layer of SiO<sub>2</sub> and polysilicon is applied on the surface of the silicon wafer.

**Step 7:** The source, gate, and drain regions are created using the diffusion process.

**Step 8:** The oxide and nitride layer is again deposited.

**Step 9:** The metal layer of aluminum is deposited on the surface of wafer including the contact holes. It also fills the cut holes.

**Step 10:** A protective glass layer is deposited on the transistor in the last step.

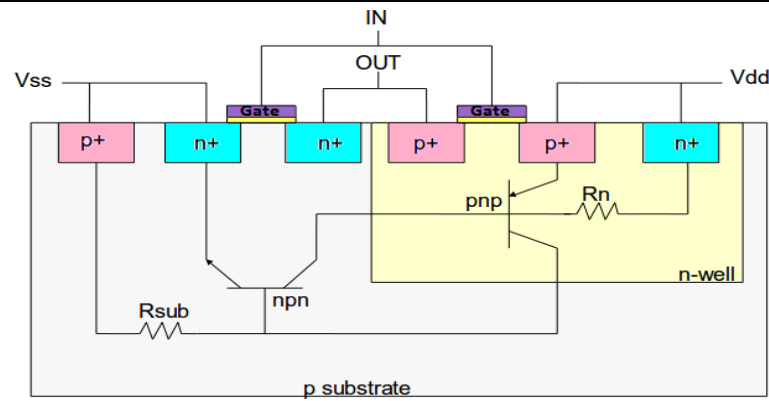
.....

### 5. CMOS fabrication by SOI technology:-

The SOI (Silicon-on-Insulator) process involves several steps to create a thin layer of silicon on top of an insulating substrate, typically silicon dioxide. The following are the typical steps involved in the SOI process:

1. **Substrate Preparation:** The first step in the SOI process is to prepare the substrate, which involves cleaning and polishing the surface to ensure that it is free of contaminants and defects.
2. **Buried Oxide Formation:** The next step is to create a layer of silicon dioxide on the surface of the substrate. This is typically done using a process called thermal oxidation, which involves heating the substrate in the presence of oxygen.
3. **Silicon Layer Deposition:** Once the silicon dioxide layer has been formed, a layer of silicon is deposited on top of it. This is typically done using a process called chemical vapor deposition (CVD), which involves heating a gas that contains silicon to deposit a layer of silicon on the substrate.
4. **Silicon Layer Thinning:** The next step is to thin the silicon layer to the desired thickness. This is typically done using a process called ion implantation, which involves bombarding the silicon with ions to create a thin, uniform layer.
5. **Annealing:** Once the silicon layer has been thinned, the substrate is heated in a process called annealing. This step helps to repair any defects in the silicon layer and improves the overall quality of the SOI substrate.
6. **Device Fabrication:** The final step in the SOI process is to fabricate devices on the SOI substrate. This typically involves using lithography and etching techniques to create patterns in the silicon layer, which can then be used to create transistors, diodes, and other electronic components.

Overall, the SOI process involves several steps to create a thin layer of silicon on top of an insulating substrate. While the process is more complex and expensive than traditional bulk silicon processing, it offers several advantages in terms of improved device performance and radiation hardness.



**Advantages:**

1. **Reduced Parasitic Capacitance:** In traditional bulk silicon processing, parasitic capacitance can reduce device performance. SOI reduces parasitic capacitance because the insulating layer isolates the silicon from the substrate, reducing interference and allowing for faster operation.
2. **Improved Transistor Performance:** SOI technology improves transistor performance because it reduces the effects of device-to-device interactions. The thin insulating layer and the substrate reduce the number of charged particles near the transistor channel, allowing for faster switching and lower leakage currents.
3. **Radiation Hardness:** SOI is also more radiation-hard than traditional bulk silicon because of its reduced parasitic capacitance and improved transistor performance. This makes SOI technology an attractive option for applications such as military and aerospace.
4. **Reduced Power Consumption:** The reduced parasitic capacitance and improved transistor performance in SOI technology can lead to reduced power consumption in devices.

**Disadvantages:**

1. **High Cost:** The SOI process is more expensive than traditional bulk silicon processing because it involves additional manufacturing steps and requires specialized equipment.
2. **Thermal Issues:** The thin insulating layer in SOI can cause thermal issues, such as localized heating, which can reduce device reliability and performance.
3. **Substrate Stress:** The insulating layer in SOI can create stress in the substrate, which can cause defects in the silicon layer and reduce device performance.
4. **Design Complexity:** SOI technology requires specialized design considerations, such as the need for floating-body effects and parasitic bipolar transistors.



## 6. Dynamic and Static Power

Dynamic Power Reduction	Static Power Reduction
Reduces power consumption when the circuit is switching or changing state	Reduces power consumption when the circuit is in a steady state
Achieved through techniques such as clock gating, power gating, and dynamic voltage and frequency scaling (DVFS)	Achieved through techniques such as reducing supply voltage, threshold voltage scaling, and transistor sizing
Dynamic power reduction techniques typically result in larger power savings	Static power reduction techniques typically result in smaller power savings compared to dynamic techniques
Suitable for applications where the circuit operates with high activity factors	Suitable for applications where the circuit operates with low activity factors
May result in a decrease in circuit performance due to the need for extra circuitry for control and management of the dynamic power reduction techniques	May not affect circuit performance or may result in a small increase in circuit performance
Examples of dynamic power reduction techniques include clock gating, power gating, and dynamic voltage and frequency scaling (DVFS)	Examples of static power reduction techniques include reducing supply voltage, threshold voltage scaling, and transistor sizing

Overall, dynamic power reduction techniques focus on reducing power consumption during the switching or changing of circuit states, while static power reduction techniques focus on reducing power consumption during steady-state operation. Both types of power reduction techniques have their own advantages and disadvantages, and the choice of which technique to use depends on the specific requirements of the application.

The formulas for dynamic power and static power are as follows:

**Dynamic Power = 0.5 x Capacitance x Voltage<sup>2</sup> x Frequency**

Static Power = Leakage Current x Supply Voltage

where:

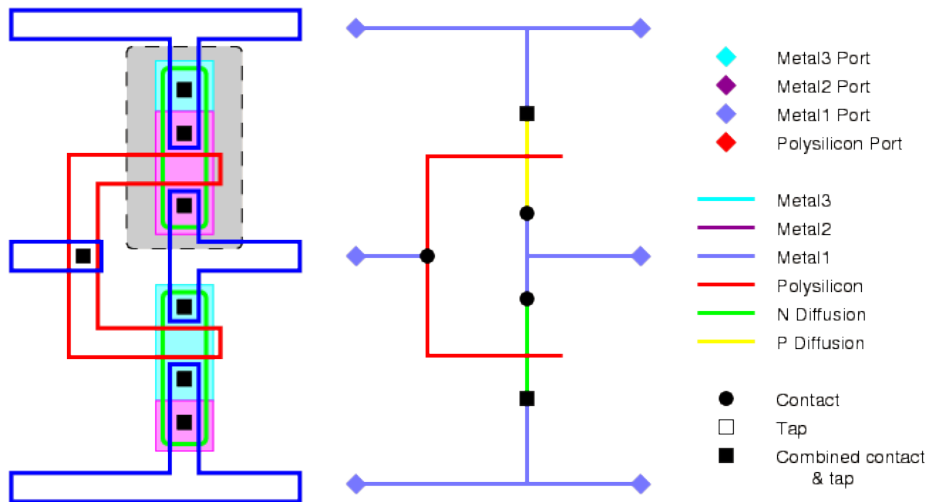
- Capacitance is the effective capacitance of the circuit (including wires, transistors, and other components)
- Voltage is the operating voltage of the circuit
- Frequency is the operating frequency of the circuit
- Leakage Current is the current that flows through a transistor when it is supposed to be turned off
- Supply Voltage is the voltage supplied to the circuit.

Dynamic power reduction techniques focus on reducing the capacitance, voltage, and frequency of the circuit, while static power reduction techniques focus on reducing the leakage current and supply voltage of the circuit. By reducing these parameters, the power consumption of the circuit can be reduced, resulting in energy savings and increased battery life in portable devices.

\*\*\*\*\*

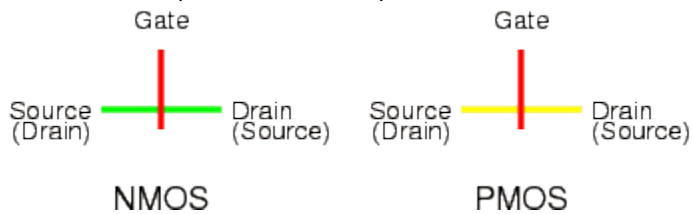
## 7. Stick diagram and layout

- Mask Layout and Stick Diagram for a CMOS Inverter



- Transistors

A transistor exists where a polysilicon stick crosses either an N diffusion stick (NMOS transistor) or a P diffusion stick (PMOS transistor).

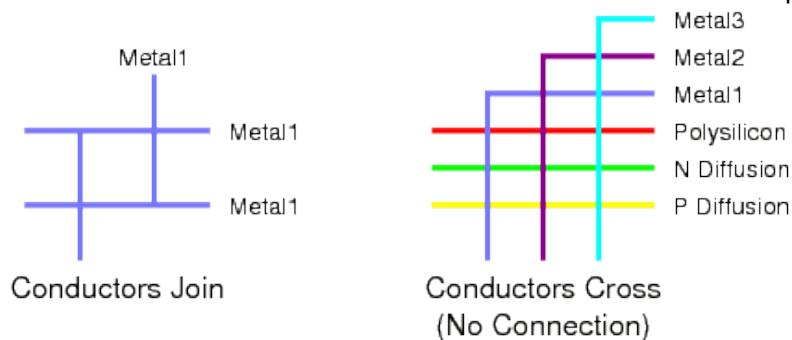


Note that there is no difference in the construction of a transistor source and a transistor drain. The source is determined as the source of conductors (electrons for NMOS / holes for PMOS) when current flows through the channel. In some pass transistor circuits, the source and drain may swap over during use.

- Implied Connections and Crossovers

Where two sticks of the same colour meet or cross there is always a connection.

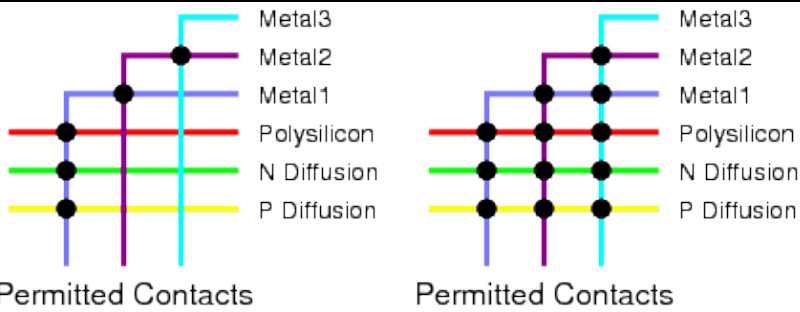
Where two sticks of different colours meet or cross there is no implied connection.



Note that N and P diffusions may not cross each other. Where poly crosses diffusion we have a transistor (see above).

- Contacts

A connection may be explicitly defined using a filled black circle. In the general case a connection is permitted where the mask layers will be separated by just one layer of insulator (through which a "contact cut" may be defined). Thus P diffusion may connect to Metal1 but not directly to Metal2.

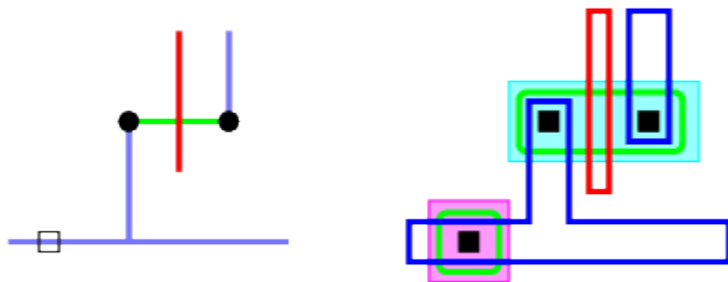


(Assuming no stacked contacts) (Assuming stacked contacts)

In a process where stacked contacts are permitted, we may draw a contact between non-adjacent conductors; e.g. between Poly and Metal3, in which case the connection to intermediate layers (Metal1 and Metal2) is implied.

- **Taps**

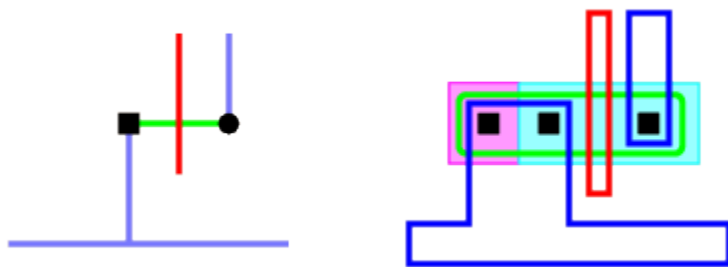
The tap represents a connection to something we can't see; either the N-Well (not shown on our stick diagram) or the wafer substrate. A tap is defined using an unfilled black square. Here there will be only one conductor crossing the square (Metal1 power or ground rail).



An N-Well Tap is inferred where the connection is from a power rail while a Substrate Tap is inferred where the connection is from a ground rail.

- **Combined Contacts & Taps**

We can often save space by using a combined contact and tap. Here the tap shares the same Active Area as the contact. A combined contact and tap is defined using a filled black square in place of the source contact (filled black circle).



A combined contact and tap can only be used where the end of a diffusion stick coincides with a contact to the power or ground rail.

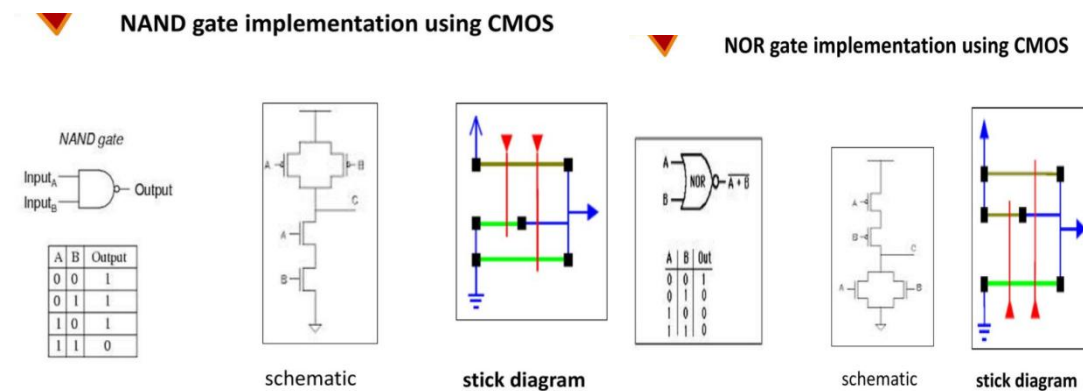
- **Stick Diagram Colour Code**

P Diffusion	: Yellow/Brown		
N Diffusion	: Green		
Polysilicon	: Red		
Metal1	: Blue		
Metal2	: Purple/Magenta		
Metal3	: Cyan/Turquoise		
Contacts & Taps	: Black		

**Layout rules:**

1. Stick diagram is a simple schematic representation of the layout of an integrated circuit. It shows the relative positions and sizes of the different components and interconnects on the chip. A stick diagram can be drawn by hand or using a computer-aided design (CAD) tool.
2. The basic components of the MOS technology used in integrated circuits are the substrate, gate oxide, polysilicon gate electrode, and metal interconnects. The stick diagram shows the position and size of each of these components in the layout.
3. Layout rules are guidelines that specify the minimum dimensions and spacing requirements for various elements of the layout, such as the width and spacing of the metal interconnects, the minimum feature size of the transistor gate electrode, and the spacing between adjacent transistors.
4. The layout rules ensure that the final layout can be reliably fabricated using the available manufacturing technology. The rules also ensure the reliability and performance of the integrated circuit.
5. When creating a stick diagram, the designer should follow the layout rules to ensure that the final layout is manufacturable and reliable. The stick diagram can then be used as a basis for creating a more detailed layout.
6. Once the stick diagram is complete, the designer can use a CAD tool to create a more detailed layout. The CAD tool will automatically enforce the layout rules to ensure that the final layout is manufacturable and reliable.

## 8. Draw the stick diagram and layout of inverter, NAND and NOR gates

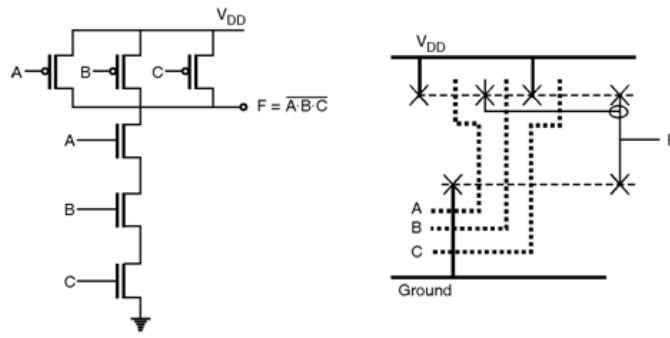


### CMOS-Layout-Design

Layout of Logic gates:

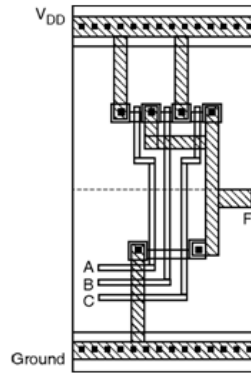
#### Three Input NAND Gate :

Figure below shows, the schematic, stick diagram and layout of three input NAND gate.



(a) Schematic

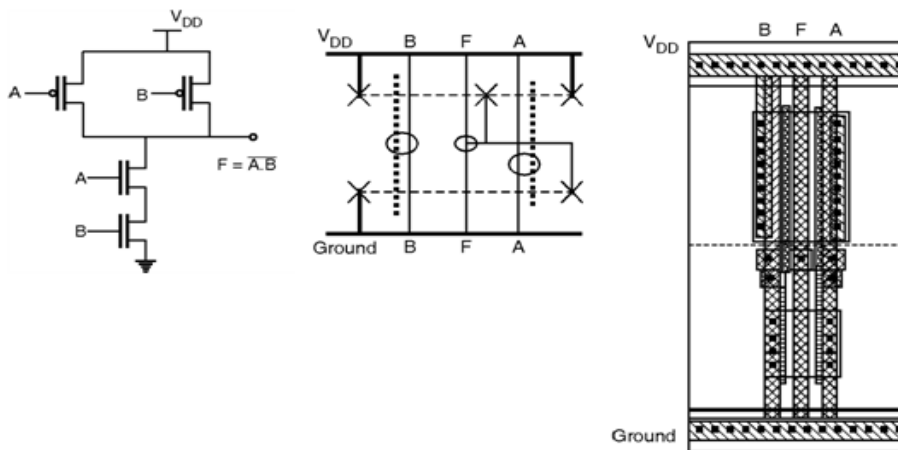
(b) Stick diagram



(c) Layout

**Two Input NAND Gate :**

Figure below shows the schematic, stick diagram and layout of two input NAND gate implemented using complementary CMOS logic.



(a) Schematic

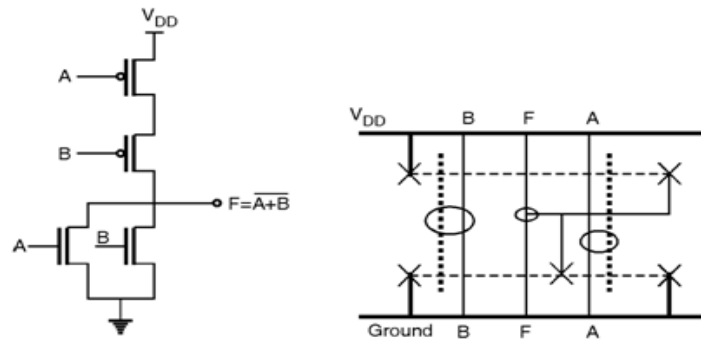
(b) Stick diagram

(c) Layout

**Two Input NOR Gate :**

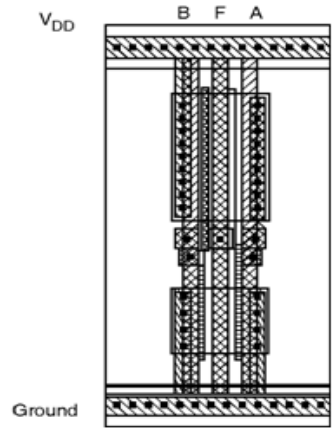
Figure below shows the schematic, stick diagram and layout of two input NOR gate implemented using complementary CMOS logic.





(a) Schematic

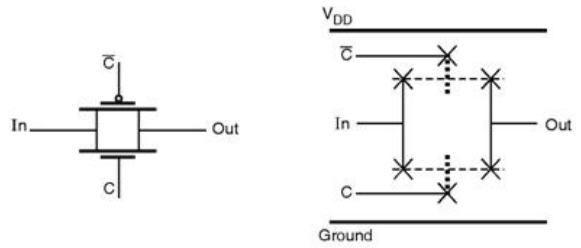
(b) Stick diagram



(c) Layout

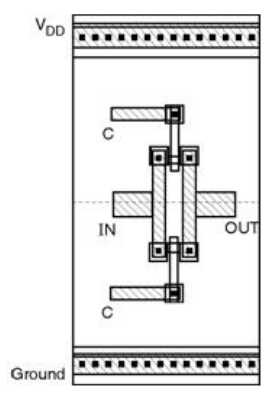
**Transmission Gate :**

Figure below shows the schematic, stick diagram and layout of the transmission gate.



(a) Schematic

(b) Stick diagram



(c) Layout



## 9. CMOS Lambda design rules

CMOS Lambda design rules are a set of guidelines for the physical layout of complementary metal-oxide-semiconductor (CMOS) circuits. The term "lambda" refers to the minimum feature size of the process, which is typically represented as a multiple of the wavelength of light used in photolithography. For example, a 0.18 micron process might be referred to as a "0.18 micron Lambda" process, where the feature size is approximately one-half of the wavelength of the light used in photolithography.

Here are some common CMOS Lambda design rules:

1. Minimum feature size: The minimum feature size, or the smallest dimension that can be reliably patterned, is typically 0.5 to 0.7 times the Lambda. For example, in a 0.18 micron Lambda process, the minimum feature size might be 0.1 to 0.13 microns.
2. Minimum spacing between features: The minimum spacing between adjacent features should be at least twice the minimum feature size. This helps to prevent short circuits between adjacent components.
3. Active area spacing: The spacing between the active areas of adjacent transistors should be at least 3 times the minimum feature size. This helps to reduce capacitive coupling between adjacent transistors.
4. Metal width and spacing: The minimum width of a metal line is typically 2 to 3 times the minimum feature size. The spacing between adjacent metal lines should be at least twice the metal width.
5. Via size: The size of a via, which connects one metal layer to another, should be at least the same as the minimum metal width.
6. Contact size: The size of a contact, which connects a metal layer to the underlying active area, should be at least the same as the minimum feature size.
7. Gate spacing: The spacing between the gate and the active area of a transistor should be at least 2 to 3 times the minimum feature size. This helps to prevent leakage between the gate and the active area.

These are just a few of the many design rules that must be followed when creating CMOS circuits. The specific rules may vary depending on the process technology and the requirements of the circuit being designed.

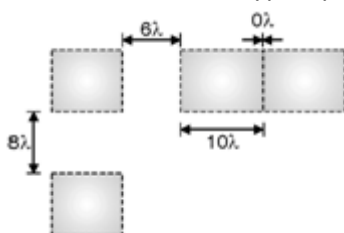
### CMOS ' $\lambda$ ' Design Rules :

The MOSIS stands for MOS Implementation Service is the IC fabrication service available to universities for layout, simulation, and test the completed designs. The MOSIS rules are scalable  $\lambda$  rules.

The MOSIS design rules are as follows :

(1) Rules for N-well as shown in Figure below.

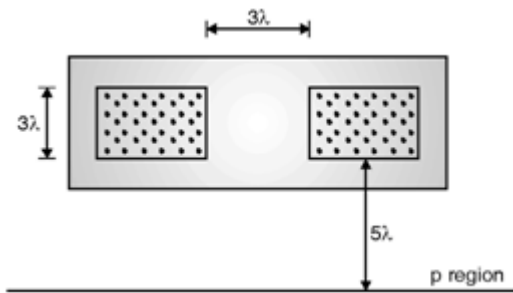
1. Minimum width =  $10\lambda$
2. Wells at same potential with spacing =  $6\lambda$
3. Wells at same potential =  $0\lambda$
4. Wells of different type, spacing =  $8\lambda$



(2) Rules for Active area shown in Figure below.

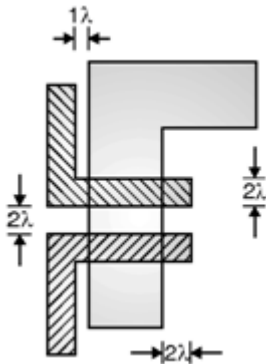
1. Minimum width =  $3\lambda$

2. Minimum spacing =  $3\lambda$
3. Source/Drain active to well edge =  $5\lambda$
4. Substrate/well contact active to well edge =  $3\lambda$



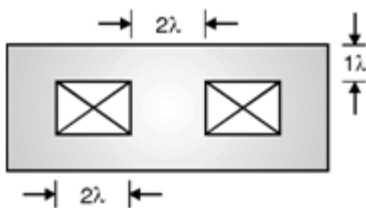
3) Rules for poly 1 as shown in Figure below.

1. Minimum width =  $2\lambda$
2. Minimum spacing =  $2\lambda$
3. Minimum gate extension of active =  $2\lambda$
4. Minimum field poly to active =  $1\lambda$



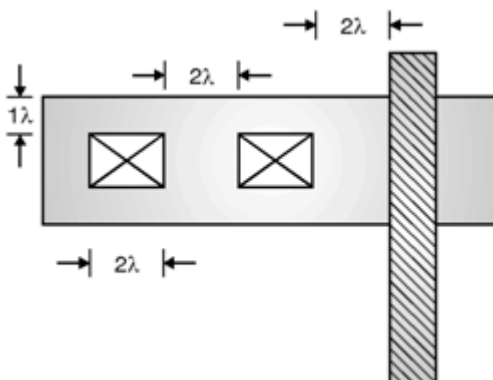
(4) Rules for contact to poly 1 as shown in Figure below.

1. Exact contact size =  $2\lambda \times 1\lambda$
2. Minimum poly 1 overlap =  $1\lambda$
3. Minimum contact spacing =  $2\lambda$



2. Minimum active overlap =  $1\lambda$  3. Minimum contact spacing =  $2\lambda$  4. Minimum spacing to gate of transistor =  $2\lambda \times (5)$

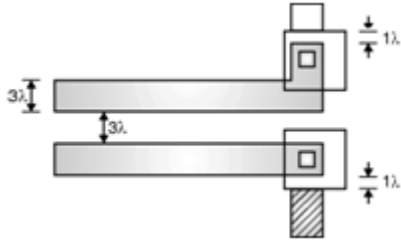
5) Rules for contact to active as shown in Figure below. 1. Exact contact size =  $2\lambda$



(6) Rules for metal 1 as shown in Figure below.

1. Minimum width =  $3\lambda$

2. Minimum spacing =  $3\lambda$
3. Minimum overlap of poly contact =  $1\lambda$
4. Minimum overlap of active contact =  $1\lambda$



(7) Rules for via 1 as shown in Figure below.

1.  $\lambda \times 1$ . Minimum size =  $2\lambda$
2. Minimum spacing =  $3\lambda$
3. Minimum overlap by metal 1 =  $1\lambda$



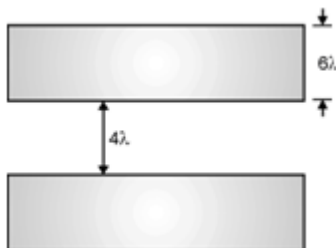
(8) Rules for metal 2 as shown in Figure below.

1. Minimum size =  $3\lambda$
2. Minimum spacing =  $4\lambda$



(9) Rules for metal 3 as shown in Figure below.

1. Minimum width =  $6\lambda$
2. Minimum spacing =  $4\lambda$



### Design Rule Check :

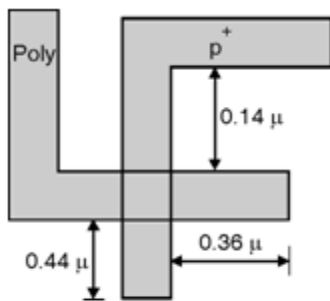
In order to ensure that none of the design rules are violated CAD tools named Design Rule Checking (DRC) is used. If DRC is not verified then it leads to the non functional design.

The layout rules are grouped in three categories that are transistor rules, contact and via rules and well and substrate contact rules.

### Transistor rules :

$m$  which is the minimum width of active layer.  $\mu m$  which is minimum width of polysilicon, whereas the width of the transistor is atleast  $0.3 \mu m$ . The transistor can be created by overlapping the the active and polysilicon layers. The minimum length of transistor equals  $0.24$

Figure below shows the layout of PMOS transistor.



PMOS transistor

Fig1-Design-Rule-Check

Contact and Via rules :

A contact forms an interconnection between metal and active or polysilicon layer whereas via forms an interconnection between two metal lines. A contact or via is formed by overlapping the two interconnecting layers and provides a contact hole filled with metal between the two.

Figure below shows the contacts and via used in layout.

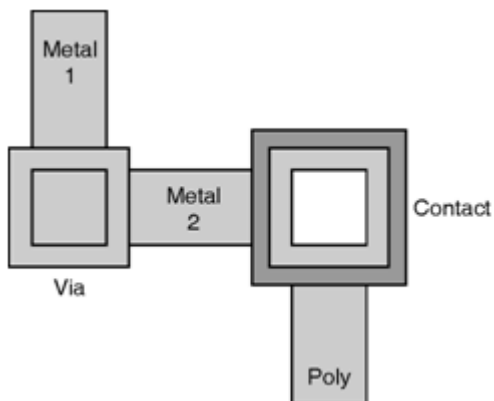


Fig1-Design-Rule-Check

Well and substrate contact rules :

For digital circuit design it is important for the well and substrate regions to be connected to the supply voltages. If this is not done then a resistive path is created between the substrate contact of the transistors and the supply rails which leads to parasitic effects such as latch up.

### CMOS-Layout-Design

Inverter Layout :

The schematic diagram of the inverter is as shown in Figure.

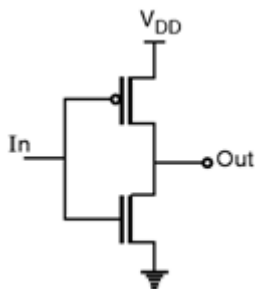


Fig1-Inverter-Layout

The stick diagram of the schematic shown in Figure.

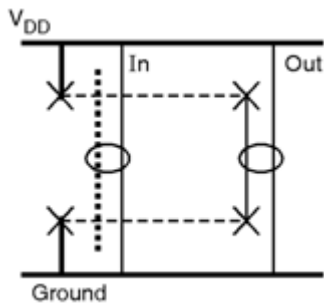


Fig2-Inverter-Layout

Here, the most important point to note is that as we change the placing of the components in the schematic the stick diagram and hence, the layout of the circuit will change accordingly. For example, if we place the components vertically the stick diagram will be vertical and if we place the components horizontally the stick diagram will be horizontal. Figure below shows the physical layout of inverter which is drawn in tanner tool.

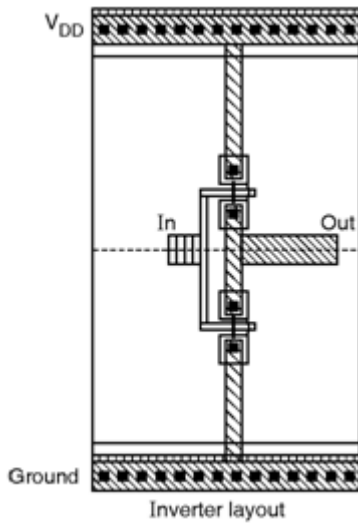


Fig2-Inverter-Layout

\*\*\*\*\*

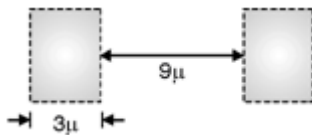
### 10. Micron-Design-Rules

Design Rules : Industry uses the micron design rules and code designs in terms of these micron dimensions. The micron design rules are as follows : ( $\mu$ )Micron

(1) Rules for N-well as shown in Figure below.

$\mu$ 1. Width = 3

$\mu$ 2. Space = 9

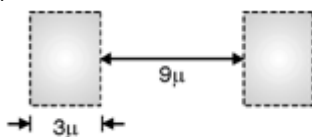


(2) Rules for active area as shown in Figure below.

$\mu$ 1. Minimum size = 3

$\mu$ 2. Minimum spacing = 3

$\mu$ 2. N+ active to N-well = 7



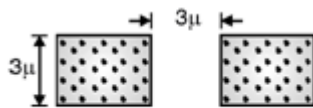
(3) Rules for poly 1 as shown in Figure below.

$\mu$ 1. Width = 2

$\mu 2$ . Spacing = 3

$\mu 3$ . Gate overlap of active = 2

$\mu 4$ . Field poly 1 to active = 1

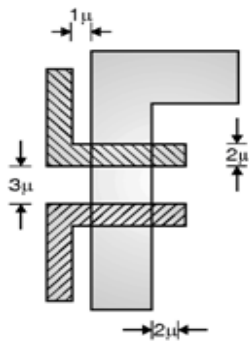


(4) Rules for contact to poly 1 as shown in Figure below.

$\mu 2 \times \mu 1$ . Exact contact size = 2

$\mu 2$ . Minimum poly overlap = 1

$\mu 3$ . Minimum contact spacing = 2



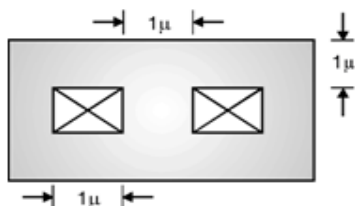
(5) Rules for contact to active as shown in Figure below.

$\mu 2 \times \mu 1$ . Exact contact size = 2

$\mu 2$ . Minimum active overlap = 1

$\mu 3$ . Minimum contact spacing = 2

$\mu 4$ . Minimum spacing to gate = 2



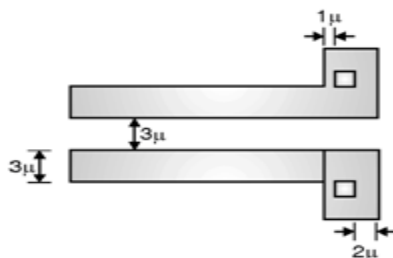
(6) Rules for metal 1 as shown in Figure below.

$\mu 1$ . Width = 3

$\mu 2$ . Spacing = 3

$\mu 3$ . Overlap of contact = 1

$\mu 4$ . Overlap of via = 2

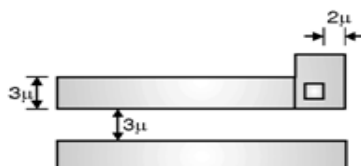


(7) Rules for metal 2 as shown in Figure below.

$\mu 1$ . Width = 3

$\mu 2$ . Space = 3

$\mu 3$ . Metal 2 overlap of via = 2





### 11. Determination of Pull up to pull down ratio.

The pull-up to pull-down ratio is an important consideration in the design of CMOS circuits, as it affects the speed, power consumption, and noise immunity of the circuit. The pull-up to pull-down ratio is defined as the ratio of the resistance of the pull-up network to the resistance of the pull-down network in a CMOS inverter.

In general, the pull-up to pull-down ratio should be as close to 1 as possible to achieve balanced rise and fall times and minimize power consumption. However, the pull-up to pull-down ratio can also affect the noise margin and stability of the circuit. For example, a smaller pull-up to pull-down ratio can result in a lower noise margin but a faster switching speed, while a larger pull-up to pull-down ratio can result in a higher noise margin but a slower switching speed.

The determination of the pull-up to pull-down ratio can be done through simulation or analytical calculations. Here are some steps for the analytical calculation method:

1. Determine the switching threshold voltage ( $V_{th}$ ) of the inverter. This is the input voltage at which the output voltage switches from high to low or low to high.
2. Calculate the resistance of the pull-up network ( $R_p$ ) and pull-down network ( $R_n$ ) using the following equations:

$$R_p = V_{dd} / (I_{leak} + I_{sat}(p) * (V_{dd} - V_{th}))$$

$$R_n = V_{dd} / (I_{leak} + I_{sat}(n) * V_{th})$$

where

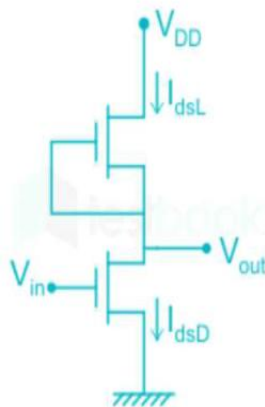
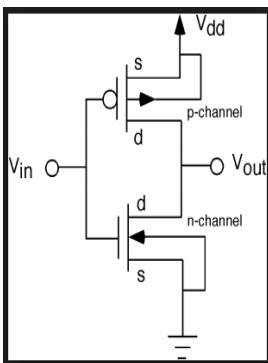
$V_{dd}$  is the supply voltage,


$I_{leak}$  is the leakage current,

$I_{sat}(p)$  and  $I_{sat}(n)$  are the saturation currents of the PMOS and NMOS transistors, respectively.

3. Calculate the pull-up to pull-down ratio ( $R_p/R_n$ ) and evaluate its effect on the noise margin, switching speed, and power consumption of the circuit.
4. Adjust the size of the transistors in the pull-up and pull-down networks to achieve the desired pull-up to pull-down ratio.

It is important to note that the determination of the pull-up to pull-down ratio is highly dependent on the specific requirements of the circuit being designed and should be optimized based on the desired trade-offs between speed, power consumption, and noise immunity.





$$\frac{Z_{P,U}}{Z_{P,D}} = \frac{(-V_{td})^2}{(V_{inv} - V_t)^2}$$





## 12. Comparing NMOS , PMOS CMOS and BiCMOS

NMOS:

- NMOS stands for "N-type Metal-Oxide-Semiconductor".
- It is a type of transistor that uses n-type semiconductor material for the channel.
- In NMOS, the gate voltage is typically higher than the source voltage to turn the transistor ON.
- NMOS is used primarily in digital circuits due to its fast switching speed and low power consumption.
- However, NMOS is not suitable for analog circuits due to its high output impedance and low noise immunity.

PMOS:

- PMOS stands for "P-type Metal-Oxide-Semiconductor".
- It is a type of transistor that uses p-type semiconductor material for the channel.
- In PMOS, the gate voltage is typically lower than the source voltage to turn the transistor ON.
- PMOS is used primarily in digital circuits due to its slow switching speed and high power consumption.
- However, PMOS is not suitable for analog circuits due to its high output impedance and low noise immunity.

CMOS:

- CMOS stands for "Complementary Metal-Oxide-Semiconductor".
- It is a type of transistor that uses both n-type and p-type semiconductor materials for the channel.
- In CMOS, both NMOS and PMOS transistors are used together in a complementary pair to achieve low power consumption and high noise immunity.
- CMOS is used extensively in digital circuits, including microprocessors, memory, and logic circuits.

BiCMOS:

- BiCMOS stands for "Bipolar Complementary Metal-Oxide-Semiconductor".
- It combines both bipolar junction transistors (BJTs) and CMOS transistors on the same chip.
- BiCMOS is used in applications that require high-speed switching, high current drive, and high voltage operation.
- BiCMOS is commonly used in mixed-signal circuits, such as ADCs, DACs, and voltage regulators.

Comparing NMOS , PMOS CMOS and BiCMOS

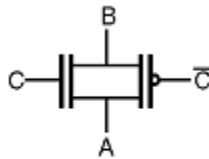
Parameter	NMOS	PMOS	CMOS	BiCMOS
<b>Device Type</b>	N-Channel MOSFET	P-Channel MOSFET	Complementary MOSFET (Both N- and P-Channel MOSFET)	Combination of Bipolar and CMOS transistors
<b>Voltage polarity</b>	Negative	Positive	Both	Both
<b>Switching speed</b>	Fast	Slow	Fast	Very Fast
<b>Power consumption</b>	Low	High	Low	Very Low
<b>Noise immunity</b>	Low	High	High	Very High
<b>Fabrication complexity</b>	Low	High	Medium	High
<b>Input impedance</b>	Low	High	Very High	Very High
<b>Output impedance</b>	High	Low	Very Low	Very Low
<b>Output swing</b>	Large	Small	Large	Large
<b>Voltage gain</b>	High	Low	Very High	High
<b>Current gain</b>	Low	High	N/A	High
<b>Application</b>	Digital	Digital	Digital	Analog and Digital

**UNIT II**

**Combinational And Sequential Circuit Design :** Pass transistor and transmission gates-inverter-NAND gates and NOR Gates for n MOS, CMOS and Bi CMOS – parity generator – multiplexers- code converters – Programmable Logic Devices (nMOS PLA and CMOS PLA) – Clocked sequential circuits – D-Latch and D- Flip-Flop –Memories (DRAM cell, SRAM Cell and Pseudo Static RAM cell) – Inverting and Non-inverting Registers – Barrel Shifter.

**1. Transmission-Gate | Pass-Transistor-Logic**

Transmission Gate Logic: The transmission gate logic is used to solve the voltage drop problem of the pass transistor logic. This technique uses the complementary properties of NMOS and PMOS transistors. NMOS devices passes a strong '0' but a weak '1' while PMOS transistors pass a strong '1' but a weak '0'. The transmission gate combines the best of the two devices by placing an NMOS transistor in parallel with a PMOS transistor as shown in Figure below. The control signals to the transmission gate C and  $\bar{C}$  are complementary to each other. The transmission gate is mainly a bi-directional switch enabled by the gate signal 'C'. When C = 1 both MOSFETs are ON and the signal pass through the gate i.e. A = B if C = 1. Whereas C = 0 makes the MOSFETs cut off creating an open circuit between nodes A and B.

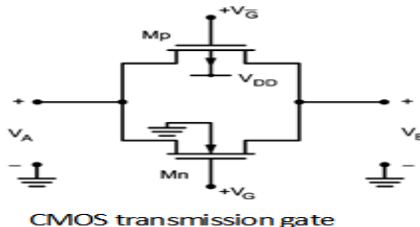


**Transmission gate**

In this section CMOS logic circuits that are based on transmission gate are implemented. This indicates the use of transmission gate to implement logic circuits.

**Basic Structure:**

The basic structure of transmission gate is shown in Figure below which consists of NMOS and PMOS transistors. Here,  $V_G$  is applied to NMOS, and  $(V_{DD}- V_G)$  applied to the PMOS.

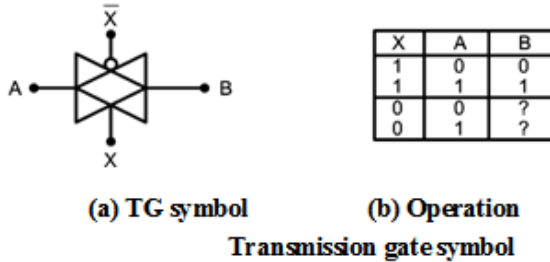


**CMOS transmission gate**

The transmission gate work voltage-controlled switch. When  $V_G$  is high, NMOS and PMOS are conducting hence switch is closed. Therefore, conduction path between left and right sides exist.

When  $V_G$  is low, then the MOSFETs are in cutoff and switch is open. Therefore, there is no direct relationship between  $V_A$  and  $V_B$ .

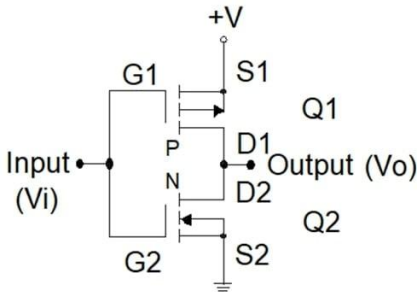
Figure below shows the symbol of transmission gate controlled by switching signals  $X$  and  $X^*$  that are applied to the gates of NMOS and PMOS respectively.



\*\*\*\*\*

**2. The CMOS Inverter or NOT Gate**

A NOT gate reverses the input logic state. *Figure 1* shows a NOT gate employing two series-connected enhancement-type MOSFETs, one n-channel (NMOS) and one p-channel (PMOS).



**Figure 1.** A CMOS NOT gate.

The input is connected to the gate terminal of the two transistors, and the output is connected to both drain terminals.

Applying +V (logic 1) to the input ( $V_i$ ), transistor Q2 is “on,” and transistor Q1 remains “off.” Under this condition, the output voltage ( $V_o$ ) is close to 0 V (logic 0).

Connecting the input to ground ( $V_i = 0$  V), transistor Q2 is “off,” and transistor Q1 is “on.” Now, the output voltage is close to +V (logic 1).

Table 1 summarizes these results.

A	Y
0	1
1	0

**Table 1.** The truth table for a NOT circuit.

### The CMOS NAND Gate

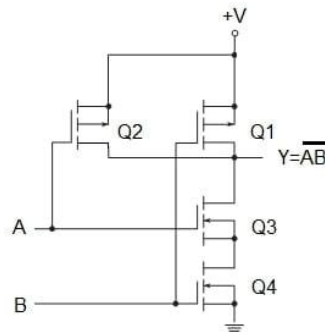
NAND denotes NOT-AND.

Table 2 shows the truth table for a NAND circuit.

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

**Table 2.** The truth table for a two-input NAND circuit.

Figure 2 shows a CMOS two-input NAND gate. P-channel transistors Q1 and Q2 are connected in parallel between +V and the output terminal. N-channel transistors Q3 and Q4 are connected in series between the output terminal and ground.



**Figure 2.** A CMOS two-input NAND gate.

With Q3 and Q4 transistors "on" and Q1 and Q2 transistors "off," the output is a logic 0. This condition happens when both inputs, A and B, are logic 1, confirming the lowest row in the above truth table.

With logic 0 in inputs A and B, Q3 and Q4 transistors are “off,” and Q1 and Q2 transistors are “on,” producing a logic 1 output. This is consistent with the first row of the truth table.

When one of the inputs is a logic “1” and the other one is a logic “0”, either Q3 is “off” and Q2 is “on” or Q4 is “off” and Q1 is “on.” The output in both cases is a logic “1,” validating the second and the third rows of the truth table.

**The NOR Gate**

NOR signifies NOT-OR.

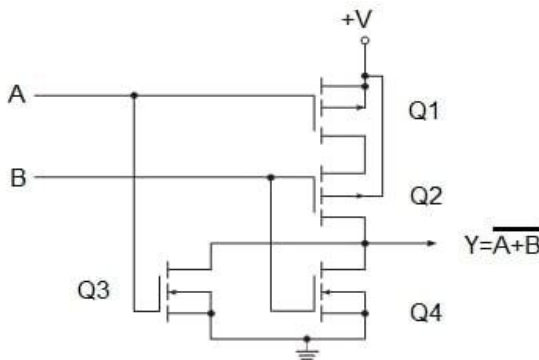
Table 3 shows the truth table for a NOR circuit.

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

**Table 3.** The truth table for a two-input NOR circuit.

The output of a NOR gate is logic 1 with logic 0 in both inputs. The outcomes for other input combinations are logic 0.

Figure 3 shows a CMOS two-input NOR gate. P-channel transistors Q1 and Q2 are connected in series between +V and the output terminal. N-channel transistors Q3 and Q4 are connected in parallel between the output and ground.



**Figure 3.** A CMOS two-input NOR gate.

When both inputs, A and B, are logic 0, Q1 and Q2 are “on,” and Q3 and Q4 are “off,” and the output is logic 1. This confirms the first row of the truth table above.

With both inputs logic 1, Q3 and Q4 are “on,” and Q1 and Q2 are “off,” producing a logic 0 output that confirms the last row of the truth table. For the two remaining input combinations, either Q1 is “off” and Q3 is “on” or Q2 is “off” and is Q4 “on”. In these cases, the output is logic 0 which is consistent with the above truth table.

\*\*\*\*\*

**3. Programmable Logic Devices (PLD)** is the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

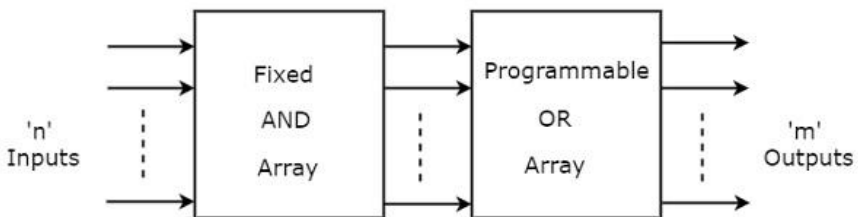
- **Programmable Read Only Memory(PROM)**
- **Programmable Array Logic(PAL)**
- **Programmable Logic Array(PLA)**

The process of entering the information into these devices is known as programming. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

**Programmable Read Only Memory PROM**

*Read Only Memory (ROM)* is a memory device, which stores the binary information permanently. That means, we can't change that stored information by any means later. If the ROM has programmable feature, then it is called as Programmable ROM PROM. The user has the flexibility to program the binary information electrically once by using PROM programmer.

**PROM** is a programmable logic device that has fixed AND array & Programmable OR array. The block diagram of PROM is shown in the following figure.



Here, the inputs of AND gates are not of programmable type. So, we have to generate  $2^n$  product terms by using  $2^n$  AND gates having  $n$  inputs each. We

can implement these product terms by using  $n \times 2^n$  decoder. So, this decoder generates 'n' min terms.

Here, the inputs of OR gates are programmable. That means, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PROM will be in the form of sum of min terms.

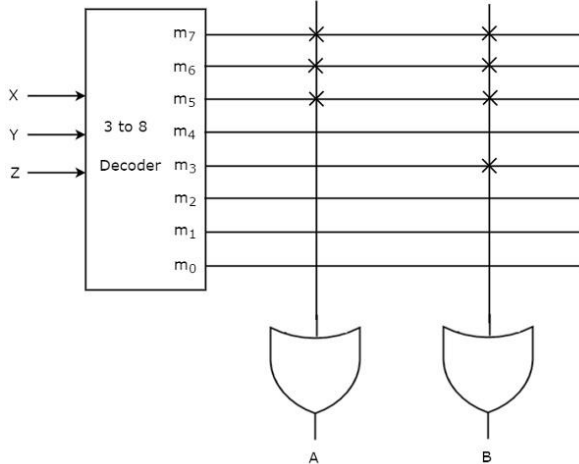
**Example**

**Let us implement the following Boolean functions using PROM.**

$$A(X,Y,Z) = \sum m(5,6,7)$$

$$B(X,Y,Z) = \sum m(3,5,6,7)$$

The given two functions are in sum of min terms form and each function is having three variables X, Y & Z. So, we require a 3 to 8 decoder and two programmable OR gates for producing these two functions. The corresponding PROM is shown in the following figure.

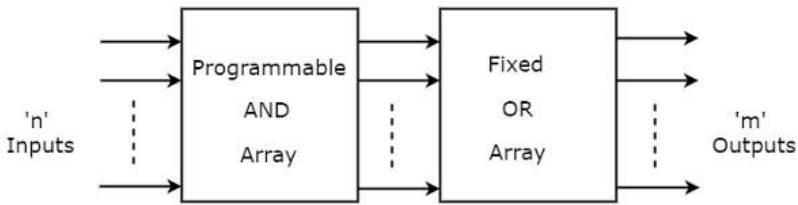


Here, 3 to 8 decoder generates eight min terms. The two programmable OR gates have the access of all these min terms. But, only the required min terms are programmed in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

**Programmable Array Logic (PAL)**

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required

product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The block diagram of PAL is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required product terms by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of sum of products form.

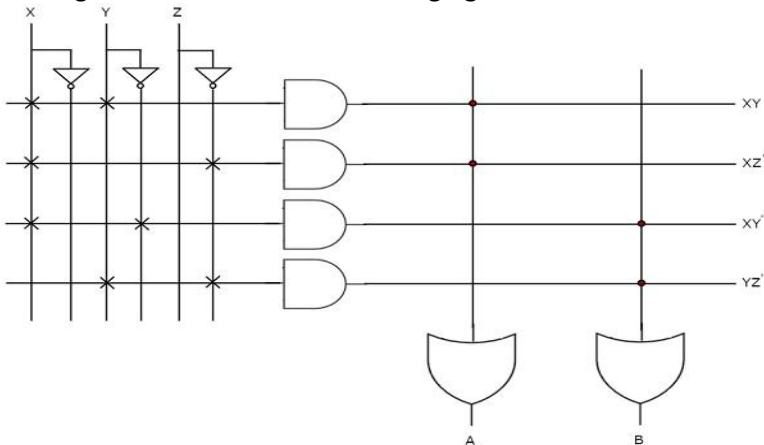
### Example

Let us implement the following Boolean functions using PAL.

$$A = XY + XZ'$$

$$B = XY' + YZ'$$

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions. The corresponding PAL is shown in the following figure.



The programmable AND gates have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X', Y, Y', Z & Z', are available at the inputs of each AND gate. So, program only the

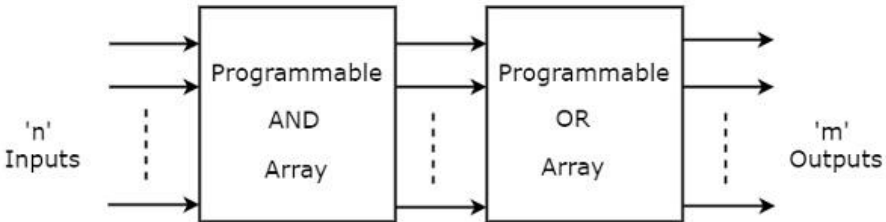


required literals in order to generate one product term by each AND gate. The symbol 'X' is used for programmable connections.

Here, the inputs of OR gates are of fixed type. So, the necessary product terms are connected to inputs of each OR gate. So that the OR gates produce the respective Boolean functions. The symbol '.' is used for fixed connections.

**Programmable Logic Array (PLA)**

PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD. The block diagram of PLA is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required product terms by using these AND gates.

Here, the inputs of OR gates are also programmable. So, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PAL will be in the form of sum of products form.

**Example**

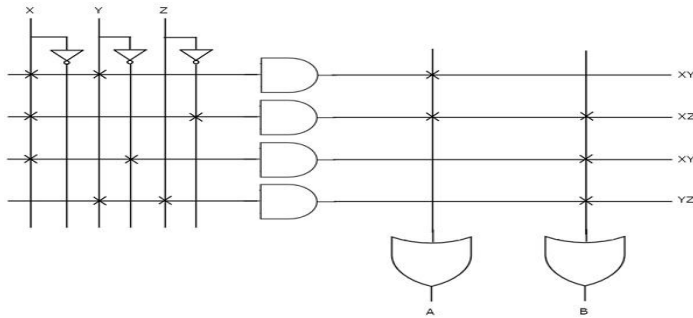
**Let us implement the following Boolean functions using PLA.**

**A=XY+XZ'**

**B=XY'+YZ+XZ'**

The given two functions are in sum of products form. The number of product terms present in the given Boolean functions A & B are two and three respectively. One product term, Z'X' is common in each function.

So, we require four programmable AND gates & two programmable OR gates for producing those two functions. The corresponding PLA is shown in the following figure.



The programmable AND gates have the access of both normal and complemented inputs of variables. In the above figure, the inputs  $X, X', Y, Y', Z$  &  $Z'$ , are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate. All these product terms are available at the inputs of each programmable OR gate. But, only program the required product terms in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

\*\*\*\*\*

#### 4. Difference between PAL and PLA

The Difference between PAL and PLA in Tabular Form mainly includes PAL and PLA full form, construction, availability, flexibility, cost, number of functions, and speed which are discussed below.

Programmable Array Logic (PAL)	Programmable Logic Array (PLA)
The full form of PAL is programmable array logic	The full form of the PLA is a programmable logic array
The construction of PAL can be done using the programmable collection of AND & OR gates	The construction of PLA can be done using the programmable collection of AND & fixed collection of OR gates.
The availability of PAL is less prolific	The availability of PLA is more
The flexibility of PAL programming is more	The flexibility of PLA is less
The cost of a PAL is expensive	The cost of PLA is middle range
The number of functions implemented in PAL is large	The number of functions implemented in PLA is limited
The speed of PAL is slow	The speed of PLA is high

\*\*\*\*\*

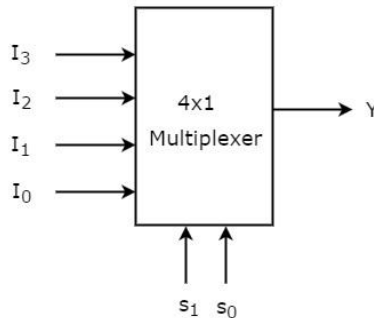
## 5. Multiplier

Multiplexer is a combinational circuit that has maximum of  $2^n$  data inputs, 'n' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are 'n' selection lines, there will be  $2^n$  possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as Mux.

### 4x1 Multiplexer

4x1 Multiplexer has four data inputs  $I_3$ ,  $I_2$ ,  $I_1$  &  $I_0$ , two selection lines  $s_1$  &  $s_0$  and one output Y. The block diagram of 4x1 Multiplexer is shown in the following figure.



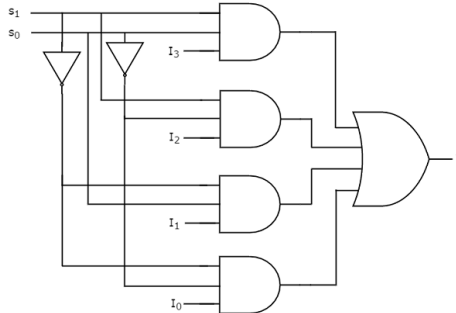
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. Truth table of 4x1 Multiplexer is shown below.

Selection Lines		Output
$S_1$	$S_0$	Y
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

From Truth table, we can directly write the Boolean function for output, Y as

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

We can implement this Boolean function using Inverters, AND gates & OR gate. The circuit diagram of 4x1 multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

### Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

#### 8x1 Multiplexer

In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

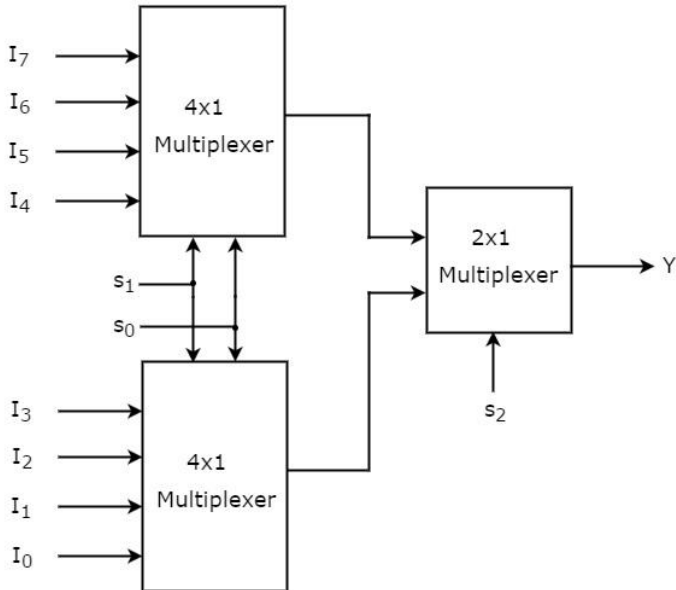
So, we require two 4x1 Multiplexers in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs  $I_7$  to  $I_0$ , three selection lines  $s_2$ ,  $s_1$  &  $s_0$  and one output  $Y$ . The Truth table of 8x1 Multiplexer is shown below.

Selection Inputs	Output
------------------	--------

$S_2$	$S_1$	$S_0$	Y
0	0	0	$I_0$
0	0	1	$I_1$
0	1	0	$I_2$
0	1	1	$I_3$
1	0	0	$I_4$
1	0	1	$I_5$
1	1	0	$I_6$
1	1	1	$I_7$

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The block diagram of 8x1 Multiplexer is shown in the following figure.



The same selection lines,  $s_1$  &  $s_0$  are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are  $I_7$  to  $I_4$  and the data inputs of lower 4x1 Multiplexer are  $I_3$  to  $I_0$ . Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines,  $s_1$  &  $s_0$ .

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other selection line,  $s_2$  is applied to 2x1 Multiplexer.

- If  $s_2$  is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs  $I_3$  to  $I_0$  based on the values of selection lines  $s_1$  &  $s_0$ .
- If  $s_2$  is one, then the output of 2x1 Multiplexer will be one of the 4 inputs  $I_7$  to  $I_4$  based on the values of selection lines  $s_1$  &  $s_0$ .

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

\*\*\*\*\*

**Example: Implementation of given function using 8 to 1 multiplexer**

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

**Solution.**

Total number of variable  $n = 4$  (A,B,C,D)

Number of select lines:  $n-1= 3$  (B, C, D)

The given function has 4 variable, so 16 possible minterms (0 – 15) are entered in the implementation table.

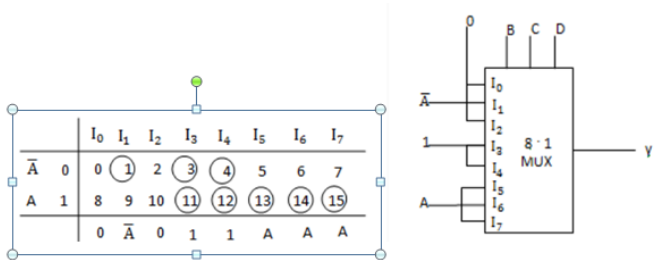
All the minterms are divided into 2 groups

The first group (0-7) minterms are entered in the first row (Variable A =0)

The second group (8–15) minterms are entered in the second row (Variable A= 1)

Circle the minterm number as per function, which you have to implement (in this case it's 1,3,4,11,12,13,14,15)

Find out the multiplexer input as per above given steps.



**Example**

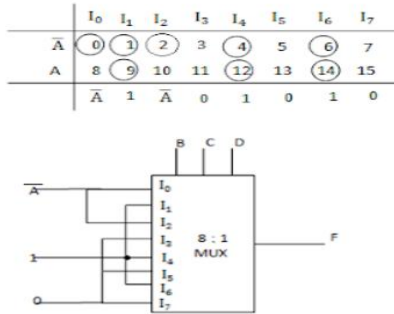
**Implement the following Boolean function using 8 : 1 MUX**

$$F(A,B,C,D) = \Sigma m(0,1,2,4,6,9,12,14)$$

**Solution.**

Select lines are B, C and D

Follow all the steps as per above points.



**Example**

**Implement the following Boolean function with 8 : 1 multiplexer**

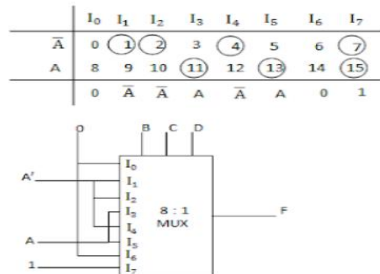
$$F(A,B,C,D) = \prod M(0,3,5,6,8,9,10,12,14)$$

**Solution**

The given maxterms are inverted to obtain minterms. From the minterms, we can implement the above Boolean function by using 8 : 1

multiplexer. Select lines are B, C and D, the input variable is A.

$$F(A,B,C,D) = \Sigma m(1,2,4,7,11,13,15)$$



**Example**

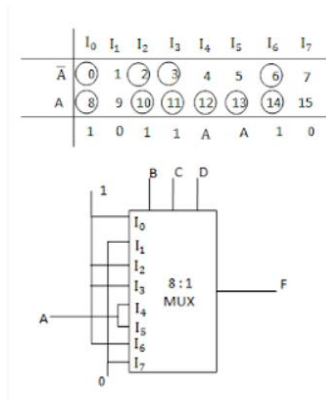
.....

Implement the following Boolean function with 8 : 1 multiplexer

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

Solution.

The Boolean function has three don't care conditions which can be treated as either 0's or 1's. In this example don't care condition is consider as



### 3. Code converter

- Binary to BCD Conversion
- BCD to Binary Conversion
- BCD to Excess-3
- Excess-3 to BCD

(a) Binary to BCD Conversion



Decimal Number	Binary code (Input)				BCD code (Output)				
	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	0	1

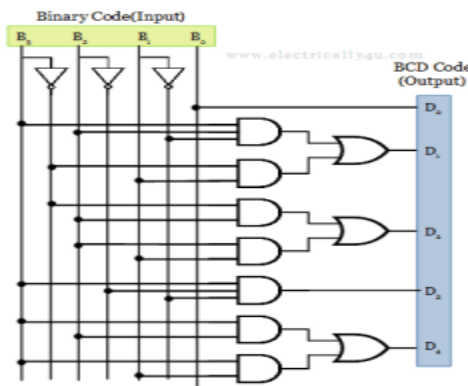
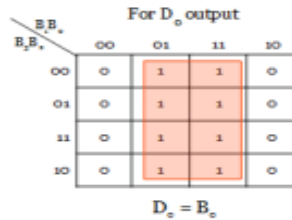
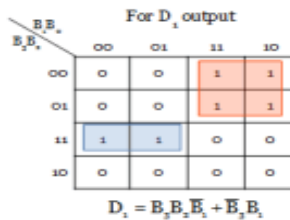
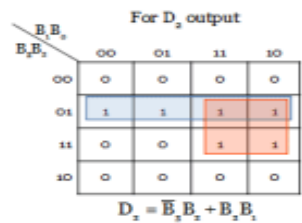
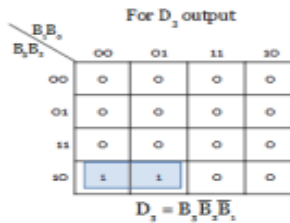
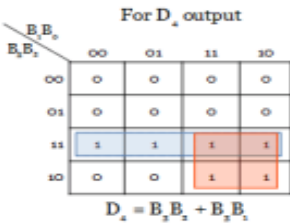
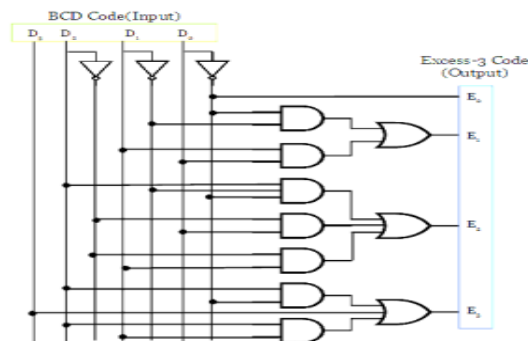
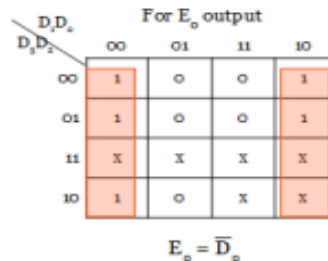
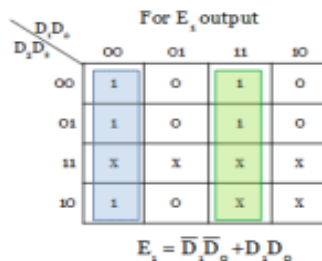
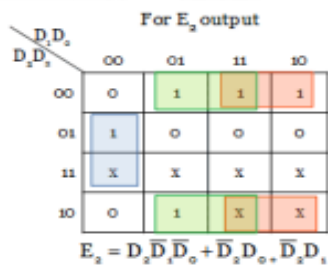
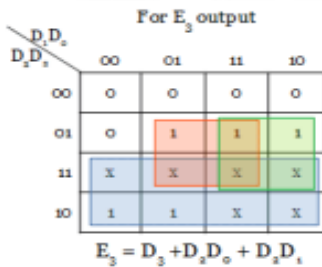


Figure 2.6.2 Binary to BCD Code Converter

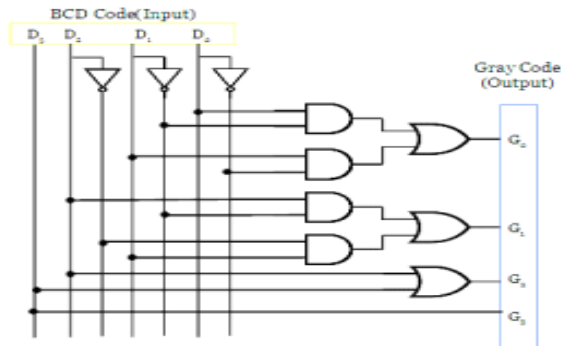
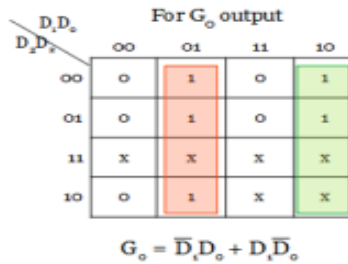
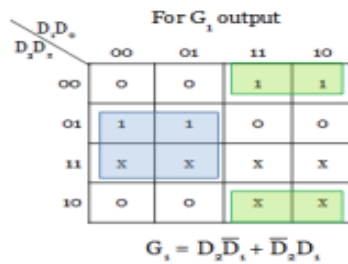
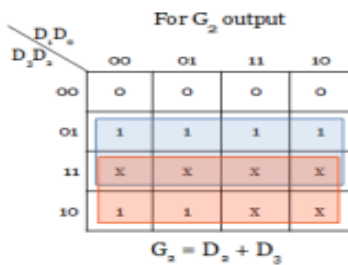
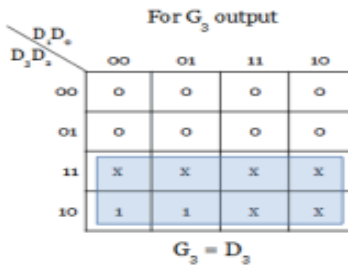
(b) BCD to Excess-3

Decimal Number	BCD code (Input)				Excess-3 code (Output)			
	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	E <sub>3</sub>	E <sub>2</sub>	E <sub>1</sub>	E <sub>0</sub>
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

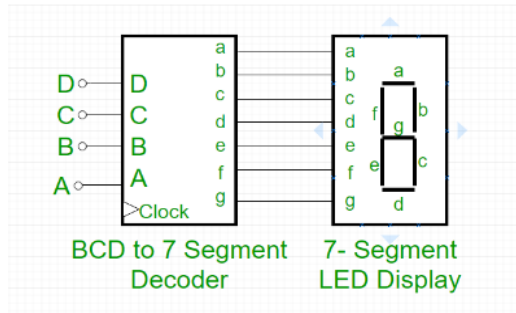


(c) BCD to Gray Code

Decimal Number	BCD code (Input)				Gray code (Output)			
	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	G <sub>3</sub>	G <sub>2</sub>	G <sub>1</sub>	G <sub>0</sub>
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1



## BCD To 7 segments



Truth Table – For common cathode type BCD to seven segment decoder:

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

\*\*\*\*\*

## 6. Parity Generator and Parity Check

### Parity Generator

It is combinational circuit that accepts an n-1 bit data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is called 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.

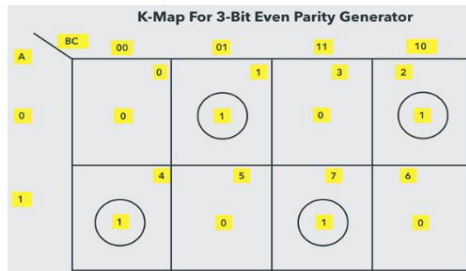
### Even Parity Generator

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 circuit 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.

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

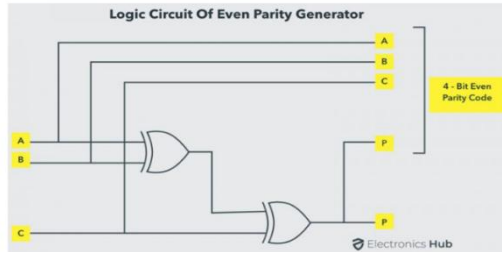


From the above truth table, the simplified expression of the parity bit can be written as

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

The above expression can be implemented by using two Ex-OR gates. 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.



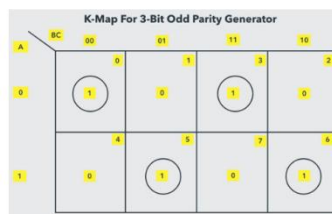
### Odd Parity Generator

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.

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



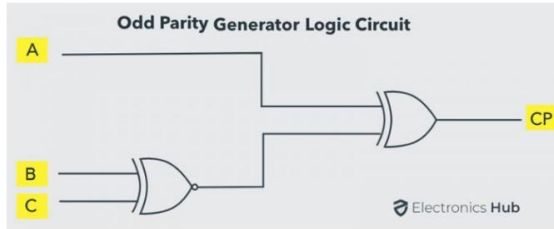
The output parity bit expression for this generator circuit is obtained as

$$P = A \oplus (\overline{B \oplus C})$$

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.



### 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.

### Even Parity Checker

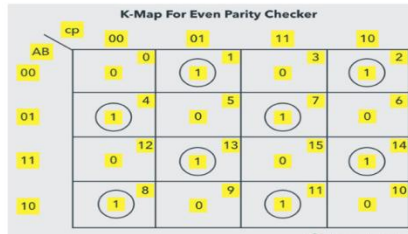
Consider that three input message along with even parity bit is generated at the transmitting end. These 4 bits are applied as input to the parity checker circuit, which checks the possibility of error on the data. Since the data is transmitted with even parity, four bits received at circuit must have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

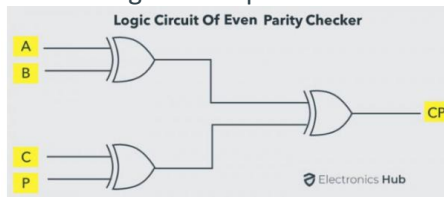
4-bit received message				Parity error check $C_p$
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

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



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

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.



### Odd Parity Checker

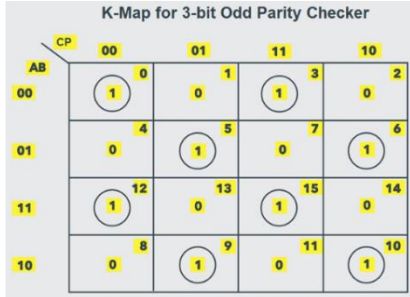
Consider that a three bit message along with odd parity bit is transmitted at the transmitting end. Odd parity checker circuit receives these 4 bits and checks whether any error are present in the data.

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.

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.

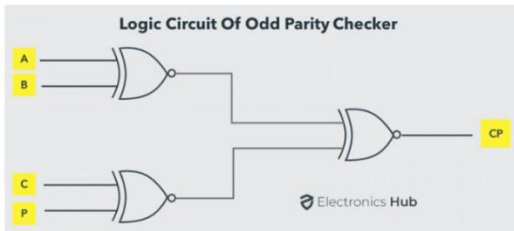


After simplification, the final expression for the PEC is obtained as

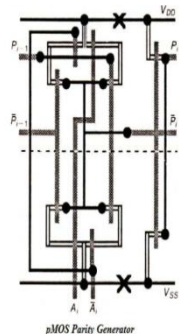
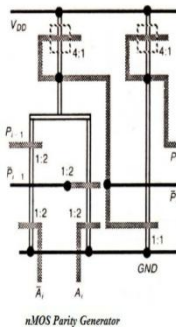
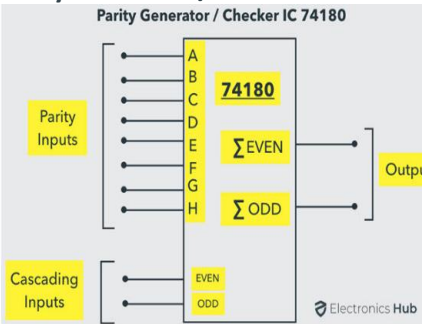
$$PEC = (A \text{ Ex-NOR } B) \text{ Ex-NOR } (C \text{ Ex-NOR } P)$$

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

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



**Parity Generator/Checker ICs**



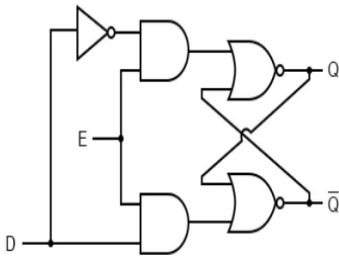
This IC consists of eight parity inputs from A through H and two cascading inputs. There are two outputs even sum and odd sum. In implementing generator or checker circuits, unused parity bits must be tied to logic zero and the cascading inputs must not be equal.

\*\*\*\*\*

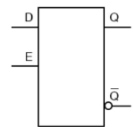
## 7. D Latch And D Flip flop

### D-LATCH

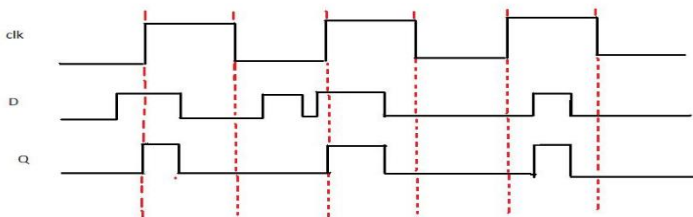
Latch is an electronic device that can be used to store one bit of information. The D latch is used to capture, or 'latch' the logic level which is present on the Data line when the clock input is high. If the data on the D line changes state while the clock pulse is high, then the output, Q, follows the input, D. When the CLK input falls to logic 0, the last state of the D input is trapped and held in the latch.



E	D	Q	$\bar{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0



### Timing diagram



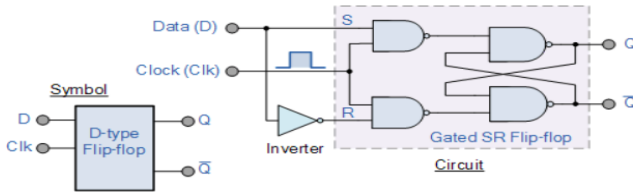
From the timing diagram it is clear that the output Q's waveform resembles that of input D's waveform when the clock is high whereas when the clock is low Q retains the previous value of D (the value before clock dropped down to 0)

### D FLIP FLOP

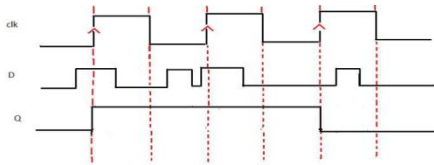
The working of D flip flop is similar to the D latch except that the output of D Flip Flop takes the state of the D input at the moment of a positive edge at the clock pin (or negative edge if the clock input is active low) and delays it by one clock cycle. That's why, it is commonly known as a delay flip flop. The D FlipFlop can be interpreted as a delay line or zero order hold. The advantage of the D flip-flop over the D-type "transparent latch" is that the signal on the D

input pin is captured the moment the flip-flop is clocked, and subsequent changes on the D input will be ignored until the next clock event.

**D-type Flip-Flop Circuit**



**Timing diagram**



**Truth Table for the D-type Flip Flop**

Clk	D	Q	$\bar{Q}$	Description
$\downarrow \approx 0$	X	Q	$\bar{Q}$	Memory no change
$\uparrow \approx 1$	0	0	1	Reset Q $\approx$ 0
$\uparrow \approx 1$	1	1	0	Set Q $\approx$ 1

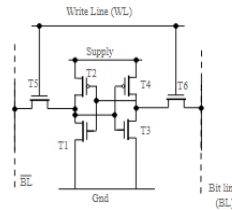
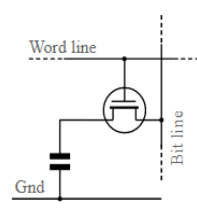
From the timing diagram it is clear that the output Q changes only at the positive edge. At each positive edge the output Q becomes equal to the input D at that instant and this value of Q is held until the next positive edge

D-Latch	D-Flip Flop
Level-sensitive latch	Edge-triggered flip-flop
Uses enable signal	Does not use enable signal
Can have transparent or opaque latch	Can have rising or falling edge trigger

D-Latch	D-Flip Flop
Output changes according to input when enable is high	Output changes only at the clock edge
Can be implemented using NOR or NAND gates	Can be implemented using NAND gates
Used in asynchronous circuits	Used in synchronous circuits
Has a single input and output	Has separate input, output, and clock signals
Can be used for simple storage or delay operations	Can be used for synchronization and timing operations

\*\*\*\*\*

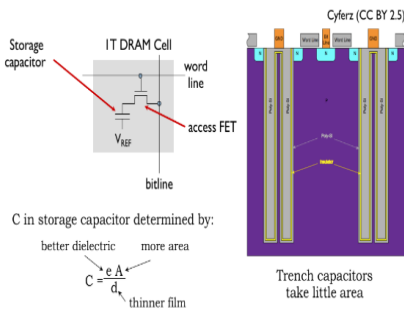
#### COMPARISON OF SRAM VS DRAM

ATTRIBUTE	STATIC RAM, SRAM	DYNAMIC RAM
<b>General</b>	SRAM uses bistable latching circuitry to store each bit. The term static differentiates it from dynamic RAM or DRAM which must be periodically refreshed.	DRAM requires periodic refreshing to retain the data
<b>Applications</b>	Generally used for Caches	Used for main computer memory
<b>Typical sizes</b>	Often up to 16 MB	Up to 16GB and beyond
<b>Cell complexity</b>		
<b>Cell density</b>	Lower than DRAM because of more complex circuitry	Much higher than SRAM because only one device is used in the cell
<b>Speed</b>	~x10 that of DRAM	Possibly around one tenth that of SRAM

## 8. Memory

Memory Type	DRAM	SRAM	PSRAM
<b>Volatility</b>	Volatile	Volatile	Volatile
<b>Storage Method</b>	Charge in capacitor	Flip-flop circuit	Charge in capacitor
<b>Cell Structure</b>	Single transistor and capacitor	Six transistors arranged in a cross-coupled configuration	Single transistor and capacitor with additional circuitry for automatic refreshing
<b>Refresh</b>	Requires constant refreshing	Does not require refreshing	Refreshes automatically
<b>Speed</b>	Slower	Faster	Faster than DRAM, slower than SRAM
<b>Power Efficiency</b>	Less power-efficient	More power-efficient	More power-efficient than DRAM, less power-efficient than SRAM
<b>Area</b>	Smaller	Larger	Intermediate

### 1T1 Dynamic RAM (DRAM) Cell

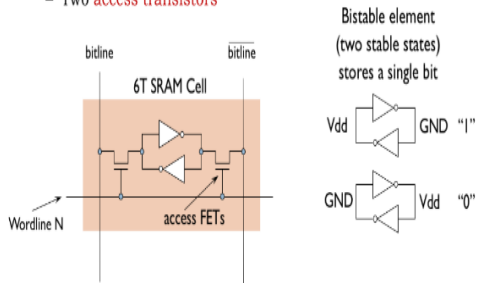


- ✓ ~20x smaller area than SRAM cell → Denser and cheaper!
- ✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)

### SRAM Cell

#### 6-MOSFET (6T) cell:

- Two CMOS inverters (4 MOSFETs) forming a **bistable element**
- Two **access transistors**



## 9. Inverting and Non inverting

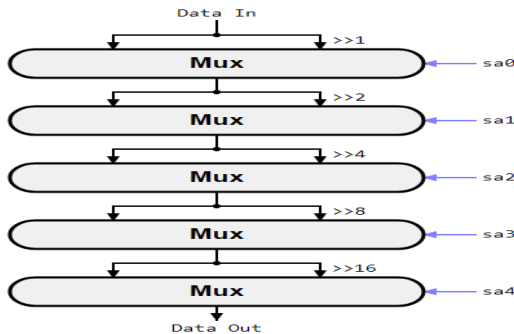
<b>Register Type</b>	<b>Inverting Register</b>	<b>Non-inverting Register</b>
Output	Inverted	Non-inverted
Functionality	Negative edge-triggered	Positive edge-triggered
Input	Active low	Active high
Feedback	Feedback from output to input is inverted	Feedback from output to input is non-inverted
Timing	Delay from input to output is propagation delay plus setup time	Delay from input to output is propagation delay only
Applications	Used in applications that require a negative edge-triggered signal, such as clocking	Used in applications that require a positive edge-triggered signal, such as data sampling

\*\*\*\*\*

## 10. Barrel Shifters

A *barrel shifter* is a logic circuit for shifting a word by a varying amount. It has a control input that specifies the number of bit positions that it shifts by.

A barrel shifter is implemented with a sequence of shift multiplexers, each shifting a word by  $2^k$  bit positions for different values of  $k$ . The diagram below shows a right-shifting barrel shifter for 32-bit words.



With more complex multiplexers and some extra circuitry for dealing with end fill options, a barrel shifter can handle all of the standard bit shift instruction in a processor's instruction set.

### Barrel shifter functionality

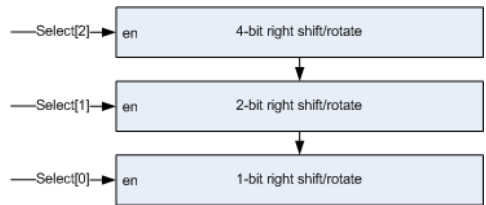
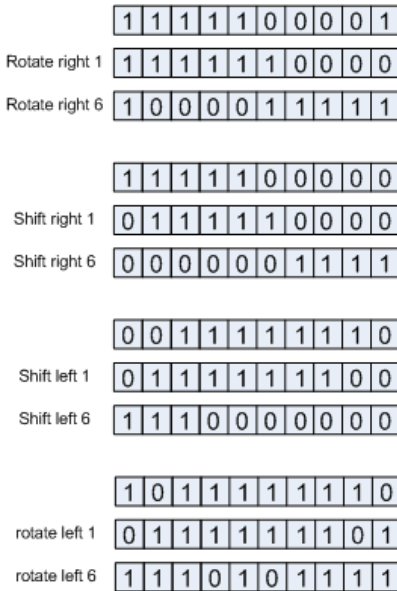
The Barrel shifter component is applicable for cases where an efficient logical shift or rotate with a selectable shift amount is required. The component supports either shift or rotate operations depending on the ROTATION parameter. When the ROTATION parameter is set to 1, the barrel shifter performs rotation and when it is set to 0, a logical shift operation is performed, shifting logical 0 in. The DIRECTION parameter determines if the barrel shifter performs a left or right shift. Setting the DIRECTION parameter to 0 would result in a left shift and setting it to 2 would result in a right shift.

The following table summarizes the operation modes of the barrel shifter:

Mode	ROTATION	DIRECTION	Description
shift left logical	0	0	Logic shift left, 0 is shifted through the rightmost (LSB) bit.
rotate left	1	0	Left rotate, the rightmost bit is shifted back in from the right.
shift right logical	0	1	Logical shift right, 0 is shifted through the leftmost (MSB) bit
rotate right	1	1	Right rotate, the rightmost bit is shifted back in from the left.

## Logarithmic shift and rotate

The shift or rotate operation is done in stages where each stage performs a shift or rotate operation of a different size. For example, a 5 bits shift operation would result in a shift of 4 and a shift of 1 where the stage that performs the shift of 2 would not do any shift. The select vector binary encoding is actually to enable the different stages of the barrel shifter.



## Logical shift operation

The logical shift operation inserts 0 value for each shift operation. The input vector is shifted in the selected direction according to the number of bits in the select indication.

## Rotate operation

The rotate operation is a shift where the bit which is shifted out of the vector MSB is inserted at its LSB.

### Rotate and shift direction

The direction of the rotate and shift operation is implemented by reversing the input and output vector. Using this method allows for the shift or rotate logic to be kept simple, performing only right shift. For a left shift, the input vector is reversed at the input, goes through the shift logic which performs a right shift according to the select input and at the output stage, it is reversed again, resulting in a left shift of the vector.



## 11. Difference between Combinational and Sequential Circuit

Key	Combinational Circuit	Sequential Circuit
<b>Definition</b>	A Combinational Circuit is a type of circuit in which the output is independent of time and only relies on the input present at that particular instant.	A Sequential circuit is a type of circuit where output not only relies on the current input but also depends on the previous output.
<b>Feedback</b>	Since output does not depend on the time instant, no feedback is required for its next output generation.	The output relies on its previous feedback so output of previous input is being transferred as feedback used with input for next output generation.
<b>Performance</b>	As the input of current instant is only required in case of Combinational circuit, it is faster and better in performance as compared to that of Sequential circuit.	Sequential circuits are comparatively slower and has low performance as compared to that of Combinational circuit.
<b>Complexity</b>	No implementation of feedback makes the combinational circuit less complex as compared to sequential circuit.	The implementation of feedback makes sequential circuit more complex as compared to combinational circuit.
<b>Elementary Blocks</b>	The elementary building blocks of a combinational circuit are its logic gates.	The building blocks of a sequential circuit are the logic gates along with flip flops.
<b>Operation</b>	Combinational circuits are mainly used for arithmetic as well as Boolean operations.	Sequential circuits are mainly used for storing data.
<b>Example</b>	Adder, Subtractor, MUX, Encoders, etc.	Flip Flops, Registers, Counters, etc.

UNIT III

**Subsystem Design :** Circuit Families – Dynamic CMOS logic, Domino CMOS logic and Pseudo NMOS logic- One bit adder- multi bit adder –Ripple carry- Carry Skip Adder-Carry Look Ahead Adder- Design of signed parallel adder-comparison of different schemes in terms of delay –Multipliers – Design of serial and parallel multipliers- different schemes and their comparison. 2’s complement array multiplication-Booth encoding.

**1. Dynamic CMOS logic**

**Dynamic logic** In static logic families the pull up and pull down networks operate concurrently. Dynamic logic on the other hand uses a sequence of precharge and conditional evaluation phases governed by the clock to realize complex logic functions.

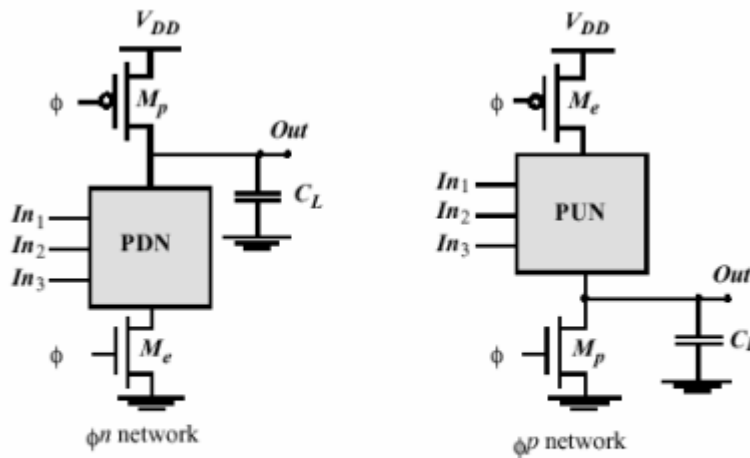
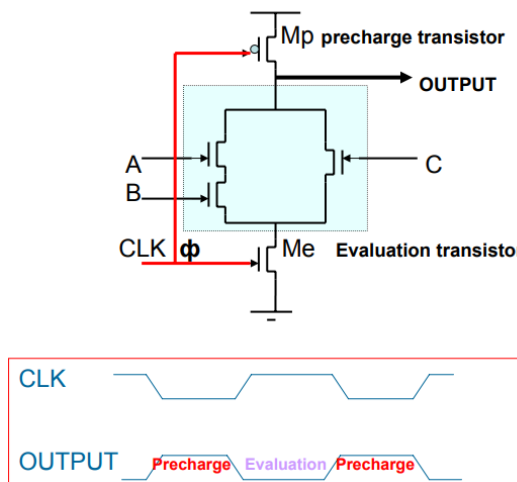


Figure 1 Dynamic logic

Figure 1 Dynamic logic A dynamic logic block is shown in Fig. 1. Both forms of Fig.1 can be used. In our analysis we will concentrate on Fig.1 n-network. The operation of the pulldown network (PDN) can be divided into two major phases. The precharge and the evaluation phase. In what mode the circuit is operating is determined by the signal  $\phi$ , the “clock” signal.



## 2. Domino CMOS logic

**Domino logic** offers a simple technique to eliminate the need for complex clocking scheme, by utilizing a single phase clock. A Domino logic module consists of a  $\phi$  n block followed by a static inverter as shown in Fig. 9. This ensures that all inputs to the next logic block are set to 0 after the precharge periods. Hence, the only possible transition during the evaluation period is 0 to 1 transition. The introduction of the static inverter has the additional advantage of the output having a low-impedance output, which increases noise immunity and drives the fan-out of the gate. The buffer furthermore reduces the capacitance of the dynamic output node by separating internal and load capacitance. The buffer itself can be optimized to drive the fan-out in an optimal way for high speed. The inverter will introduce another problem that this type of logic family is non inverting.

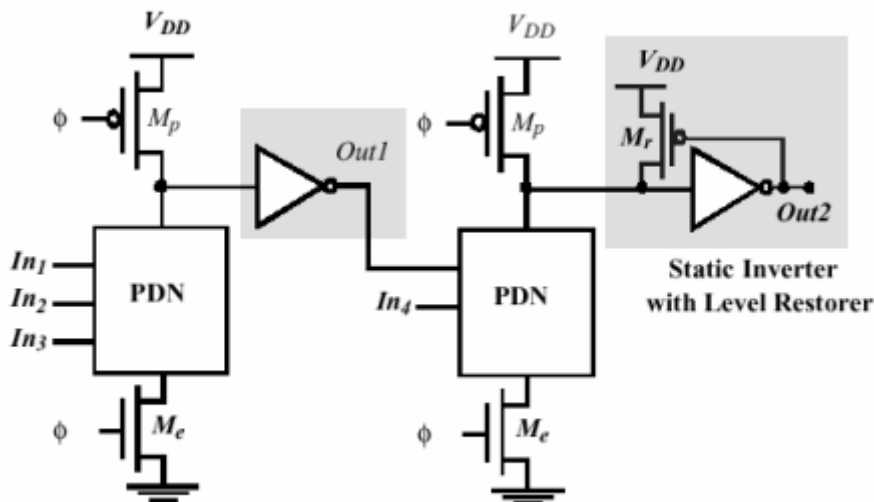


Figure 9 The block of Domino logic

Consider now the operation of a chain of Domino gates. During precharge, all inputs are set to 0. During the evaluation, the output of the first Domino block either stays at 0 or makes a 0 to 1 transition, affecting the second Domino. This effect might ripple through the whole chain, one after the other, as with a line of falling dominoes.

Domino CMOS has the following properties:

- Each gate requires  $N+4$  transistors
- Logic evaluation propagates as falling dominoes hence minimum evaluation period is determined by the logic depth.
- The nodes must be precharged during the precharge period. Total precharge time depends on size of pmos.
- Inputs must be stable (only one rising transition) during the evaluation period.
- Gates are ratio less and are non inverting. Domino gates can be made more immune to parasitic effects by adding a level-restoring transistor to the static CMOS inverter.

Even though CMOS logic gates have very low power dissipation, they have the following limitations:

1. They occupy larger area than NMOS gates.
2. Due to the larger area, they have larger capacitance.
3. Larger capacitance leads to longer delay in switching.

These limitations of the CMOS gates can be reduced by several alternative structures discussed below. These structures resemble the CMOS structure in some way; yet, they are able to reduce the chip area and hence the capacitive delay. Pseudo-NMOS logic, dynamic NMOS logic, and domino logic are some of these special CMOS structures.

### 3. Pseudo-NMOS (p-NMOS) Logic Gates

Figure 3.32 shows a pseudo-NMOS inverter (p-NMOS NOT) gate, Fig. 3.33 shows a pseudo-NMOS NAND (p-NMOS NAND) gate, and Fig. 3.34 shows a pseudo-NMOS NOR (p-NMOS NOR) gate. As shown in all these figures, there is a block of NMOS FETs, which will contain one or more NMOS transistors, as required by the structure of the gate. However, there will be only one PMOS transistor in any pseudo-NMOS logic, and this will be always grounded.

The p-NMOS circuit is a modification of NMOS circuits with DMOS loads. In p-NMOS circuits, we use a PMOS transistor, instead of the DMOS transistor, as its load. The advantages of using a PMOS load are:

- ☐ The circuit retains its basic CMOS structure. Hence, the chip area is minimum compared with the conventional CMOS structure.
- ☐ The circuit becomes compatible with CMOS devices.

The channel resistance of the pseudo-NMOS devices is higher than that of the NMOS devices. Hence, power dissipation is lower for the pseudo devices.

- ☐ Pseudo-NMOS circuits are useful in applications where the output remains in logic-1 state most of the time.

However, the pseudo circuits have the disadvantage that they possess greater propagation delay than the NMOS devices. Pseudo NMOS Logic Circuit is shown below.

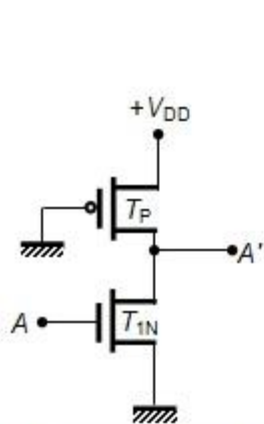


Fig. 3.32 p-NMOS NOT

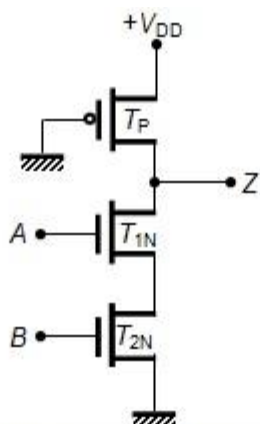


Fig. 3.33 p-NMOS NAND

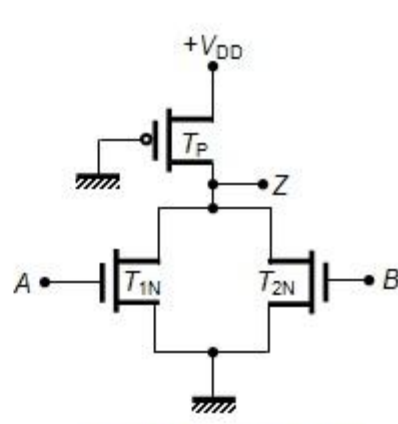


Fig. 3.34 p-NMOS NOR

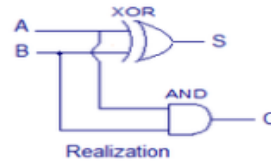
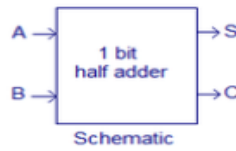
### 4. One bit adder

#### Half adder

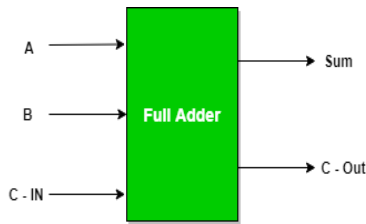
The half adder adds two single binary digits A and B. It has two outputs, sum (S) and carry (C). The carry signal represents an overflow into the next digit of a multi-digit addition. The value of the sum is  $2C + S$ . The simplest half-adder design, pictured on the right, incorporates an XOR gate for S and an AND gate for C.

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

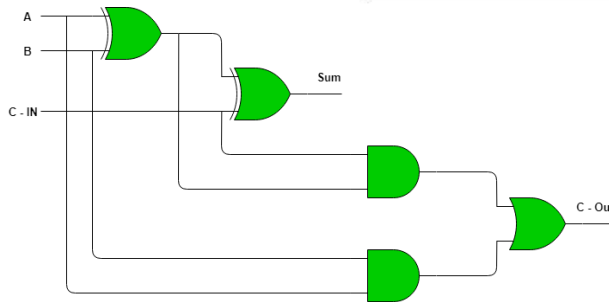
Truth table



**Full Adder** is the adder that adds three inputs and produces two outputs. The first two inputs are A and B and the third input is an input carry as C-IN. The output carry is designated as C-OUT and the normal output is designated as S which is SUM. A full adder logic is designed in such a manner that can take eight inputs together to create a byte-wide adder and cascade the carry bit from one adder to another. we use a full adder because when a carry-in bit is available, another 1-bit adder must be used since a 1-bit half-adder does not take a carry-in bit. A 1-bit full adder adds three operands and generates 2-bit results.



Inputs			Outputs	
A	B	C - IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



### 5. Differences between Dynamic CMOS logic, Domino CMOS logic, and Pseudo NMOS logic:

Feature	Dynamic CMOS logic	Domino CMOS logic	Pseudo NMOS logic
Logic gates	MOSFET transistors	MOSFET transistors	NMOS transistor and resistor
Charge storage	Capacitor	Capacitor	No charge storage
Precharge operation	Not used	Inverter chain precharges output node to high state	Not used
Clock signal requirement	Yes	Yes	No
Noise immunity	Limited	Limited	Less noise-immune
Power consumption	Low	Low	High
Circuit complexity	Moderate	Moderate	Simple
Usage in modern	Widely used	Widely used in high-	Rarely used

Feature	Dynamic CMOS logic	Domino CMOS logic	Pseudo NMOS logic
circuit design		performance applications	

**Ripple Carry Adder, Carry Skip Adder, and Carry Look Ahead Adder** are three types of adders used in digital circuits to perform binary addition.

**Ripple Carry Adder (RCA):** A ripple carry adder is the simplest form of adder that is used for adding two n-bit binary numbers. It uses multiple full adders connected in series, where the carry output of one full adder is connected to the carry input of the next full adder. The main disadvantage of this adder is that it has a long propagation delay, which increases with the number of bits to be added.

**Carry Skip Adder (CSA):** The Carry Skip Adder (also known as the Carry Bypass Adder) is an improvement over the ripple carry adder. This adder uses a series of 2:1 multiplexers to selectively skip groups of full adders when the carry input is not changing. The advantage of this adder is that it reduces the propagation delay of the adder when the carry input is constant.

**Carry Look Ahead Adder (CLA):** The Carry Look Ahead Adder (CLA) is another improvement over the ripple carry adder. It uses a set of precomputed carry signals to determine the carry output of each full adder in parallel, instead of sequentially. The advantage of this adder is that it has a constant propagation delay, regardless of the number of bits being added.

#### 6. Comparison table of the Ripple Carry Adder, Carry Skip Adder, and Carry Look Ahead Adder:

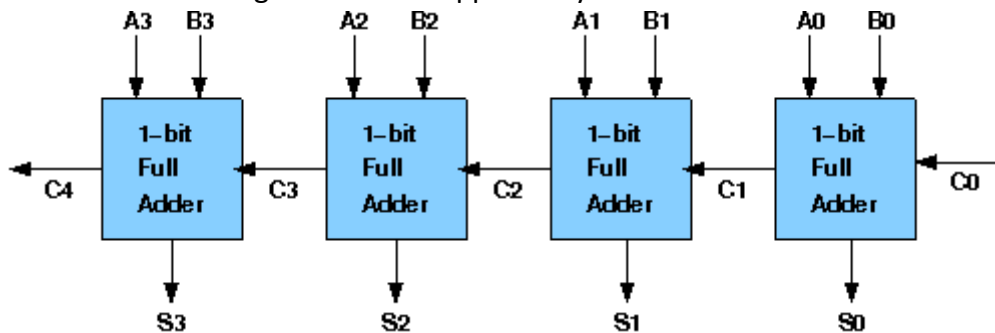
Feature	Ripple Carry Adder (RCA)	Carry Skip Adder (CSA)	Carry Look Ahead Adder (CLA)
Logic	Simple, using full adders	More complex, using full adders and multiplexers	More complex, using precomputed carry signals
Propagation delay	Increases with the number of bits	Reduces when carry input is constant	Constant
Gate delay	High	Low to moderate	Low
Hardware complexity	Simple	Moderate	High
Area required	Minimal	Larger than RCA	Largest
Power consumption	Low	Moderate	High
Application	Low-speed and low-power applications	Medium-speed and medium-power applications	High-speed and high-power applications
Cost	Low	Moderate	High
Adder speed	Slow	Faster than RCA	Fastest
Carry generation	Sequential	Concurrent	Precomputed
Carry lookahead	N/A	Short	Moderate

Feature	Ripple Carry Adder (RCA)	Carry Skip Adder (CSA)	Carry Look Ahead Adder (CLA)
time			
Critical path length	Increases with the number of bits	Increases with the number of bits	Constant
Carry lookahead adder	No	No	Yes

## 7. Ripple Carry Adders:

Arithmetic operations like addition, subtraction, multiplication, division are basic operations to be implemented in digital computers using basic gates like AND, OR, NOR, NAND etc. Among all the arithmetic operations if we can implement addition then it is easy to perform multiplication (by repeated addition), subtraction (by negating one operand) or division (repeated subtraction).

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers. Each full adder inputs a **Cin**, which is the **Cout** of the previous adder. This kind of adder is a **Ripple Carry Adder**, since each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder. The block diagram of 4-bit Ripple Carry Adder is shown here below -



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is  $31 * 2$  (for carry propagation) + 3 (for sum) = 65 gate delays.

### Design Issues :

The corresponding boolean expressions are given here to construct a ripple carry adder. In the half adder circuit the sum and carry bits are defined as

$$\text{sum} = A \oplus B$$

$$\text{carry} = AB$$

In the full adder circuit the the Sum and Carry output is defined by inputs A, B and Carryin as

$$\text{Sum} = ABC + ABC + ABC + ABC$$

$$\text{Carry} = ABC + ABC + ABC + ABC$$

Having these we could design the circuit. But, we first check to see if there are any logically equivalent statements that would lead to a more structured equivalent circuit.

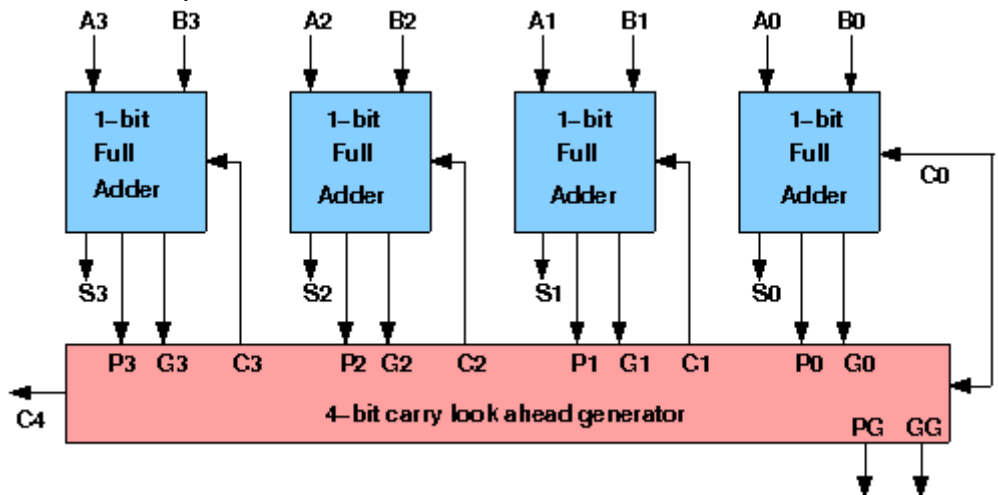
With a little algebraic manipulation, one can see that

$$\text{Sum} = ABC + ABC + ABC + ABC$$

$$\begin{aligned}
&= (AB + AB) C + (AB + AB) C \\
&= (A \oplus B) C + (A \oplus B) C \\
&= A \oplus B \oplus C \\
\text{Carry} &= ABC + ABC + ABC + ABC \\
&= AB + (AB + AB) C \\
&= AB + (A \oplus B) C
\end{aligned}$$

### 8. Carry Lookahead Adders

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be **Carry Propagator** and **Carry Generator**. The carry propagator is propagated to the next level whereas the carry generator is used to generate the output carry, regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry  $C_{in}$  to output carry  $C_{out}$  requires an AND gate and an OR gate, which constitutes two gate levels. So if there are four full adders in the parallel adder, the output carry  $C_5$  would have  $2 \times 4 = 8$  gate levels from  $C_1$  to  $C_5$ . For an n-bit parallel adder, there are  $2n$  gate levels to propagate through.

Design Issues :

The corresponding boolean expressions are given here to construct a carry lookahead adder. In the carry-lookahead circuit we need to generate the two signals carry propagator(P) and carry generator(G),

$$\begin{aligned}
P_i &= A_i \oplus B_i \\
G_i &= A_i \cdot B_i
\end{aligned}$$

The output sum and carry can be expressed as

$$\begin{aligned}
\text{Sum}_i &= P_i \oplus C_i \\
C_{i+1} &= G_i + (P_i \cdot C_i)
\end{aligned}$$

Having these we could design the circuit. We can now write the Boolean function for the carry output of each stage and substitute for each  $C_i$  its value from the previous equations:

$$\begin{aligned}
C_1 &= G_0 + P_0 \cdot C_0 \\
C_2 &= G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0 \\
C_3 &= G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 \\
C_4 &= G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0
\end{aligned}$$



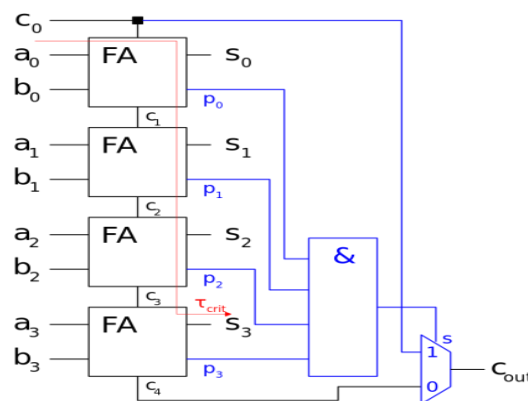
**9. A Carry Skip Adder (CSA)** is a type of digital circuit used in computer processors and other digital systems to add two binary numbers together. It is a high-speed parallel adder that can add two n-bit binary numbers in a single clock cycle.

The basic idea behind a CSA is to use a combination of carry look-ahead and carry select adders to reduce the delay caused by ripple carry. A carry look-ahead adder generates carry bits for each bit position based on the input bits, while a carry select adder selects the carry-in for each block of bits based on the carry-out from the previous block.

The CSA divides the input bits into blocks of k bits, where k is a power of 2. The carry select adder is then used to add these blocks together, and the carry look-ahead adder is used to generate the carry bits between the blocks. The carry bits are then propagated from block to block until they reach the final output.

One advantage of a CSA is that it can reduce the number of stages required to perform an addition, which can lead to faster operation. Additionally, because the carry bits are generated and propagated independently of the input bits, the CSA is well suited to parallel processing and pipelining.

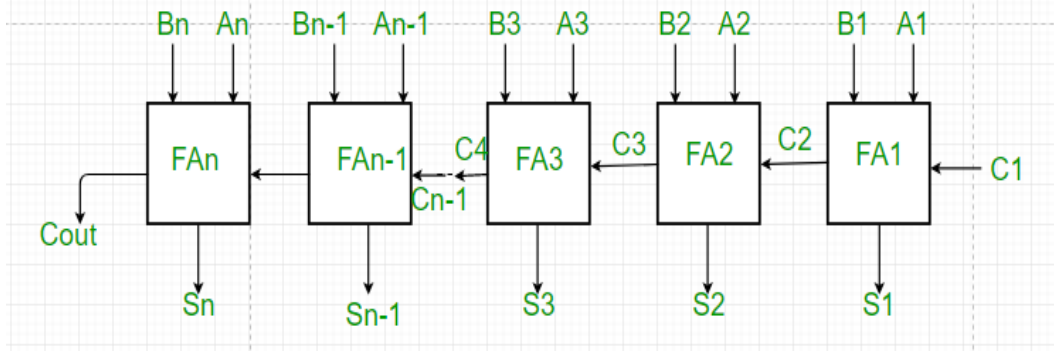
Overall, the Carry Skip Adder is a popular and efficient technique for fast addition of binary numbers in digital systems.



Full adder with additional generate and propagate signals.

## 10. Parallel Adder –

A single full adder performs the addition of two one bit numbers and an input carry. But a **Parallel Adder** is a digital circuit capable of finding the arithmetic **sum** of two binary numbers that is **greater than one bit** in length by operating on corresponding pairs of bits in parallel. It consists of **full adders connected in a chain** where the output carry from each full adder is connected to the carry input of the next higher order full adder in the chain. **A n bit parallel adder requires n full adders to perform the operation.** So for the two-bit number, two adders are needed while for four bit number, four adders are needed and so on. Parallel adders normally incorporate carry lookahead logic to ensure that carry propagation between subsequent stages of addition does not limit addition speed.



1. As shown in the figure, firstly the full adder FA1 adds A1 and B1 along with the carry C1 to generate the sum S1 (the first bit of the output sum) and the carry C2 which is connected to the next adder in chain.
2. Next, the full adder FA2 uses this carry bit C2 to add with the input bits A2 and B2 to generate the sum S2 (the second bit of the output sum) and the carry C3 which is again further connected to the next adder in chain and so on.
3. The process continues till the last full adder FAn uses the carry bit Cn to add with its input An and Bn to generate the last bit of the output along last carry bit Cout.

### 11. Comparing the different schemes in terms of delay:

Scheme	Basic Idea	Advantages	Disadvantages	Delay Characteristics
Ripple Carry Adder (RCA)	Add bits sequentially and propagate carry bits	Simple and easy to implement	Linearly increases with the number of bits being added	The delay increases significantly for large numbers of bits. The output of each stage depends on the carry-out of the previous stage, causing a ripple effect that slows down the circuit.
Carry Ahead Adder (CLA)	Generate carry bits for each bit position based on input bits	Faster than RCA, can add large numbers of bits	Circuit size and power consumption increase rapidly with number of bits	The delay is proportional to the number of bits, but it is much faster than RCA. The carry bits are generated independently of the input bits, so the circuit can operate in parallel. However, the circuit size and power consumption increase rapidly with the number of bits, making it less efficient than more advanced schemes.

Scheme	Basic Idea	Advantages	Disadvantages	Delay Characteristics
Carry Select Adder (CSA)	Divide input bits into blocks and use combination of carry look-ahead and carry select adders	Fast and efficient, can add two n-bit binary numbers in a single clock cycle	Requires additional hardware for carry select adder and carry look-ahead adder	Significantly reduced delay compared to RCA and CLA. The circuit can add two n-bit binary numbers in a single clock cycle. The carry bits are generated and propagated independently of the input bits, allowing for parallel processing and pipelining. However, it requires additional hardware for the carry select adder and carry look-ahead adder.
Carry-Save Adder (CSA)	Save carry bits in a separate register and perform addition once all numbers have been added	Efficient for adding multiple numbers	Requires additional hardware for storing and retrieving carry bits	Reduces the delay by performing addition only once all numbers have been added. The carry bits are saved in a separate register, so they do not need to be generated and propagated through the circuit. However, it requires additional hardware for storing and retrieving the carry bits, which can increase the circuit size and power consumption.
Carry-Completion Adder (CCA)	Use carry select adders and a carry completion circuit to reduce delay caused by ripple carry	Efficient and can handle large numbers of bits	Requires additional hardware for the carry completion circuit	More efficient than RCA and CLA, and faster than CSA for large numbers of bits. The carry completion circuit ensures that all the carry bits are correctly generated and propagated through the circuit. However, it requires additional hardware for the carry completion circuit.

**12. The multiplier** uses the serial-parallel method of addition to calculate the result of multiplying two 8-bit numbers as shown in figure 2.1 below. The multiplier receives the two operands A and B and outputs the result C. Operands A and B are loaded in parallel into 8-bit registers and the result C is shifted into a 16-bit register. Multiplication begins on the assertion of a START signal and once the calculation is complete a STOP signal is asserted.

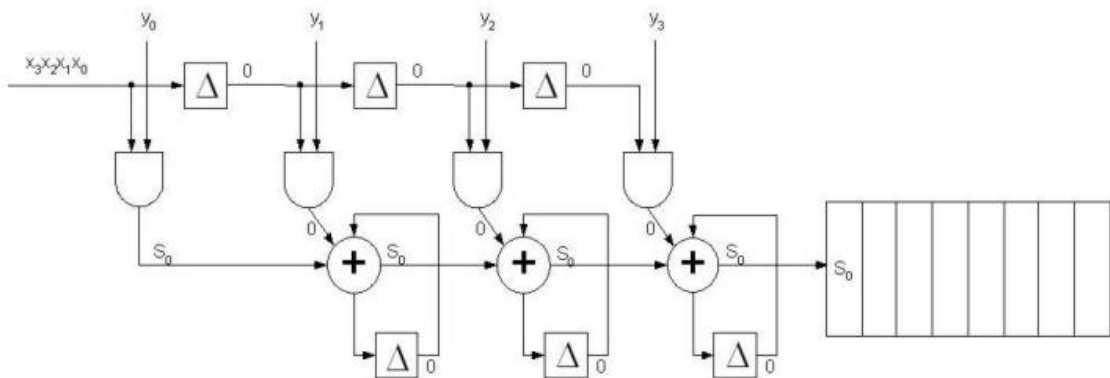


Figure 2.1 Serial-Parallel Multiplier.

The serial-parallel multiplier is based on the addition of bits in the corresponding column of the multiplication process as shown below. Each column is added in one clock cycle generating the corresponding bit. The resulting bit is then shifted into output register. Therefore the entire multiplication process for the 8 by 8-bit multiplier requires 16 clock cycles to complete the calculation.

$$\begin{array}{r}
 \phantom{a}a1a0 \\
 \times \phantom{a}b1b0 \\
 \hline
 \phantom{a}a1b0 \phantom{a}a0b0 \\
 a1b1 \phantom{a}a0b1 \\
 \hline
 a1b1 (a1b0 + a0b1) \phantom{a}a0b0
 \end{array}$$

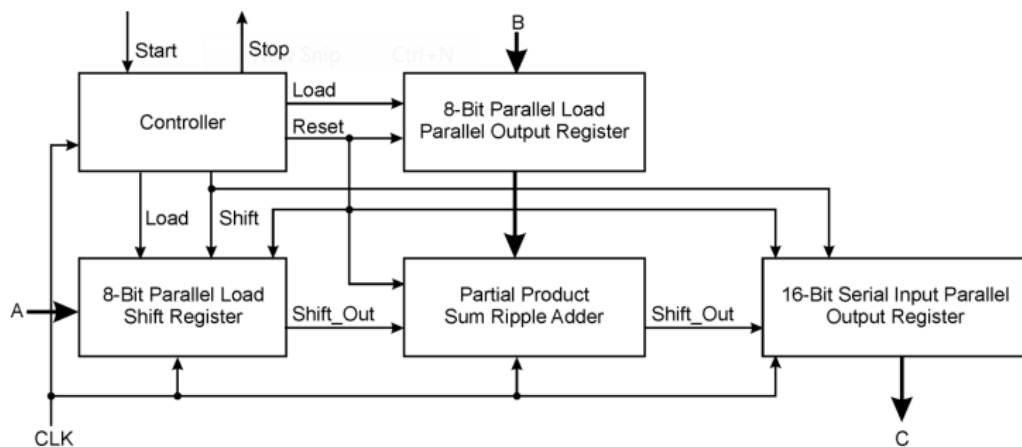


Figure 2.2 Multiplier Block Diagram.

The block diagram for the multiplier is shown in figure 2.2 below. The first operand, A, is loaded in parallel and the most significant bit is shifted out during each clock cycle. Operand B is also loaded in parallel and its value is stored in the register for the entire multiplication process. The result C is generated by shifting the added bits of each column one by one into the resultant register. Therefore register RA is a parallel load shift register, RB is a parallel load parallel output register, and RC is a serial input parallel output register.

**13. Serial/Parallel Multiplier** The general architecture of the serial/parallel multiplier is shown in the figure below. One operand is fed to the circuit in parallel while the other is serial. N partial products are formed each cycle. On successive cycles, each cycle does the addition of one column of the multiplication table of M\*N PPs. The final results are stored in the output register after N+M cycles. While the area required is N-1 for M=N. For snapshots of data transfer through this multiplier please see the course website/slides of lecture.

Multiplying two numbers represented in two's complement requires a series of bitwise operations and additions. The following steps outline the general process:

1. Convert the two's complement numbers to their decimal equivalents.
2. Multiply the decimal values of the two numbers.
3. Convert the decimal result back to two's complement format.

Here's an example of multiplying two 4-bit two's complement numbers (in binary) using this method:

<pre> 1101 (-3) x 0110 (6) ----- 0000 (0) + 1101 (-3) ----- 101110 (-18) </pre>	<pre> Binary representation of 5 is: 0 1 0 1 1's Complement of 5 is:      1 0 1 0  2's Complement of 5 is: (1's Complement + 1) i.e.                         1 0 1 0 (1's Compliment)                           + 1                         <u>1 0 1 1</u> (2's Complement i.e. -5) </pre>
---	--

In this example, we're multiplying -3 and 6, which in two's complement representation are 1101 and 0110, respectively.

We begin by multiplying the decimal equivalents of these numbers (which are -3 and 6), which gives us -18. We then convert -18 to binary in 4-bit two's complement format, which is 101110.

**14. Booth encoding** is a technique used in digital circuit design to reduce the number of partial products that need to be added together when performing multiplication.

In binary multiplication, each bit of the multiplicand is multiplied with each bit of the multiplier, resulting in multiple partial products. Booth encoding is a method that reduces the number of partial products by representing adjacent groups of 1's in the multiplier as a single value.

For example, suppose we want to multiply the 4-bit binary numbers 0110 and 1011. Using the standard multiplication method, we would obtain the following partial products:

```

0110
x1011
-----
0110
0000
1100

```

**0110**

100010

Using Booth encoding, we can represent the multiplier as a sequence of 0's, 1's, and -1's, where adjacent groups of 1's are combined into a single -1. For example, the multiplier 1011 would be encoded as -101. Then, we perform the multiplication using the following steps:

1. Write the multiplicand in the first column and the encoded multiplier in the second column, shifting the multiplier to the right by one position for each column.

```

0110
0
-1
0
1

```

2. Starting from the rightmost column, examine each pair of bits in the multiplier. If the pair is 01, add the multiplicand to a running total; if the pair is 10, subtract the multiplicand from the running total.

```

0110
0
-1
0
1

```

**0110**

3. Shift the multiplicand and the encoded multiplier one position to the right and repeat step 2 until all columns have been processed.

```

0110
0
-1
0
1

```

**0110**

**0011**

**0000**

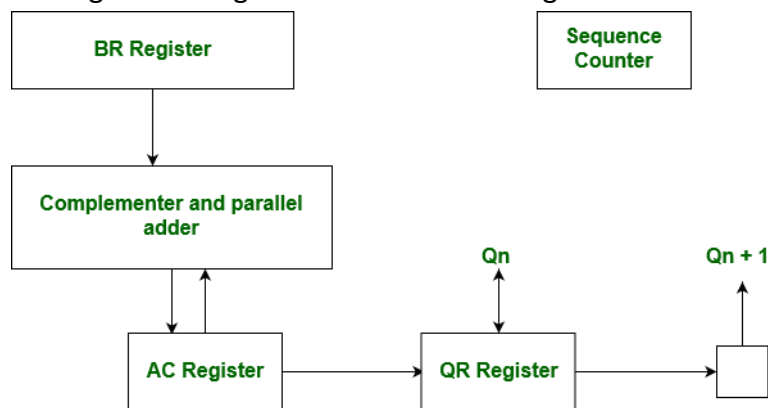
100010

As you can see, using Booth encoding reduced the number of partial products from four to three, resulting in fewer additions and faster multiplication.

**Booth algorithm** gives a procedure for **multiplying binary integers** in signed 2's complement representation **in efficient way**, i.e., less number of additions/subtractions required. It operates on the fact that strings of 0's in the multiplier require no addition but just shifting and a string of 1's in the multiplier from bit weight  $2^k$  to weight  $2^m$  can be treated as  $2^{(k+1)}$  to  $2^m$ . As in all multiplication schemes, booth algorithm requires examination **of the multiplier bits** and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to following rules:

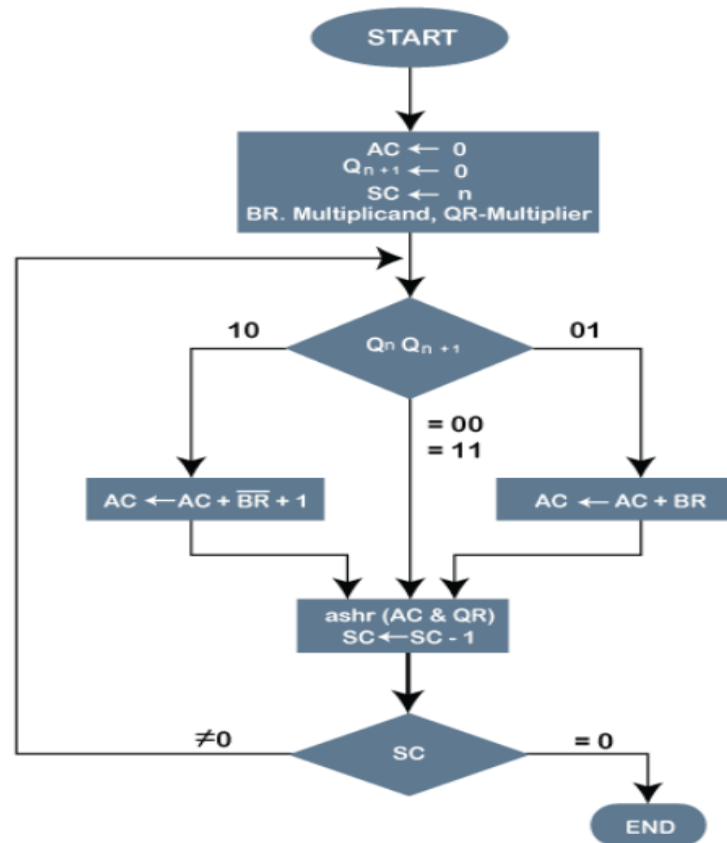
1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous '1') in a string of 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

**Hardware Implementation of Booths Algorithm** – The hardware implementation of the booth algorithm requires the register configuration shown in the figure below.



#### Booth's Algorithm Flowchart –

We name the register as A, B and Q, AC, BR and QR respectively.  $Q_n$  designates the least significant bit of multiplier in the register QR. An extra flip-flop  $Q_{n+1}$  is appended to QR to facilitate a double inspection of the multiplier. The flowchart for the booth algorithm is shown below.



*Flow chart of Booth's Algorithm.*

AC and the appended bit  $Q_{n+1}$  are initially cleared to 0 and the sequence SC is set to a number  $n$  equal to the number of bits in the multiplier. The two bits of the multiplier in  $Q_n$  and  $Q_{n+1}$  are inspected. If the two bits are equal to 10, it means that the first 1 in a string has been encountered. This requires subtraction of the multiplicand from the partial product in AC. If the 2 bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the 2 numbers that are added always have a opposite signs, a condition that excludes an overflow. The next step is to shift right the partial product and the multiplier (including  $Q_{n+1}$ ). This is an arithmetic shift right (ashr) operation which AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated  $n$  times. Product of negative numbers is important, while multiplying negative numbers we need to find 2's complement of the number to change its sign, because it's easier to add instead of performing binary subtraction. Product of two negative number is demonstrated below along with 2's complement.

**Example** – A numerical example of booth's algorithm is shown below for  $n = 4$ . It shows the step by step multiplication of -5 and -7.

BR = -5 = 1011,

BR' = 0100, ← 1's Complement (change the values 0 to 1 and 1 to 0)

BR'+1 = 0101 ← 2's Complement (add 1 to the Binary value obtained after 1's complement)

QR = -7 = 1001 ← 2's Complement of 0111 (7 = 0111 in Binary)

The explanation of first step is as follows:  $Q_{n+1}$



AC = 0000, QR = 1001,  $Q_{n+1} = 0$ , SC = 4

$Q_n Q_{n+1} = 10$

So, we do  $AC + (BR)'+1$ , which gives AC = 0101

On right shifting AC and QR, we get

AC = 0010, QR = 1100 and  $Q_{n+1} = 1$

OPERATION	AC	QR	$Q_{n+1}$	SC
	0000	1001	0	4
AC + BR' + 1	0101	1001	0	
ASHR	0010	1100	1	3
AC + BR	1101	1100	1	
ASHR	1110	1110	0	2
ASHR	1111	0111	0	1
AC + BR' + 1	0100	0111	0	
ASHR	0010	0011	1	0

Product is calculated as follows:

Product = AC QR

Product = 0010 0011 = 35

#### Advantages:

**Faster than traditional multiplication:** Booth's algorithm is faster than traditional multiplication methods, requiring fewer steps to produce the same result.

**Efficient for signed numbers:** The algorithm is designed specifically for multiplying signed binary numbers, making it a more efficient method for multiplication of signed numbers than traditional methods.

**Lower hardware requirement:** The algorithm requires fewer hardware resources than traditional multiplication methods, making it more suitable for applications with limited hardware resources.

**Widely used in hardware:** Booth's algorithm is widely used in hardware implementations of multiplication operations, including digital signal processors, microprocessors, and FPGAs.

#### Disadvantages:

**Complex to understand:** The algorithm is more complex to understand and implement than traditional multiplication methods.

**Limited applicability:** The algorithm is only applicable for multiplication of signed binary numbers, and cannot be used for multiplication of unsigned numbers or numbers in other formats without additional modifications.

**Higher latency:** The algorithm requires multiple iterations to calculate the result of a single multiplication operation, which increases the latency or delay in the calculation of the result.

**Higher power consumption:** The algorithm consumes more power compared to traditional multiplication methods, especially for larger inputs.

## VLSI Design

### UNIT IV

**CMOS Testing** : Need for testing- Test Procedure, Design for Testability – Ad Hoc Testing – Scan-Based Test-Boundary-Scan Design – Built-in-Self-Test(BIST)- Test-Pattern Generation – Fault Models – Automatic Test Pattern Generation – Fault Simulation.

#### 1. Need of Testing in CMOS

**Testing** is particularly important in the context of Complementary Metal-Oxide-Semiconductor (CMOS) technology because of the following reasons:

1. **High integration and complexity:** CMOS technology enables the integration of millions of transistors on a single chip, resulting in highly complex electronic circuits. The complexity increases the likelihood of defects or faults in the circuits, making testing essential to ensure reliable operation.
2. **Fabrication variability:** Fabrication of CMOS circuits involves a complex set of processes, and each process can have variations that can lead to defects or faults in the final product. Testing helps to identify and correct these defects or faults.
3. **Performance optimization:** CMOS circuits are designed for specific performance criteria, such as speed, power consumption, and noise immunity. Testing is essential to ensure that the circuits meet these performance criteria and operate reliably under different conditions.
4. **Safety-critical applications:** CMOS circuits are widely used in safety-critical applications, such as medical devices, automotive systems, and aerospace systems. Testing is critical to ensuring that these circuits operate safely and reliably under all conditions.
5. **Time-to-market:** Testing is essential to reducing the time-to-market for CMOS products. Testing early in the development process helps to identify and correct defects or faults, which reduces the time and cost required for later testing and corrections.
6. **Product quality:** Testing is essential to ensuring high product quality for CMOS circuits. Quality testing helps to identify defects or faults that could affect product performance or reliability, ensuring that the product meets customer expectations.

#### 2. Test Procedure in CMOS

The test procedure in CMOS technology typically involves several stages of testing, including wafer testing, package testing, and final testing. Here is a general overview of the test procedure in CMOS technology:

1. **Wafer testing:** Wafer testing is the first stage of testing in CMOS technology. It involves testing the individual dies on the wafer before they are packaged. The tests performed during wafer testing include functional testing, parametric testing, and defect testing.

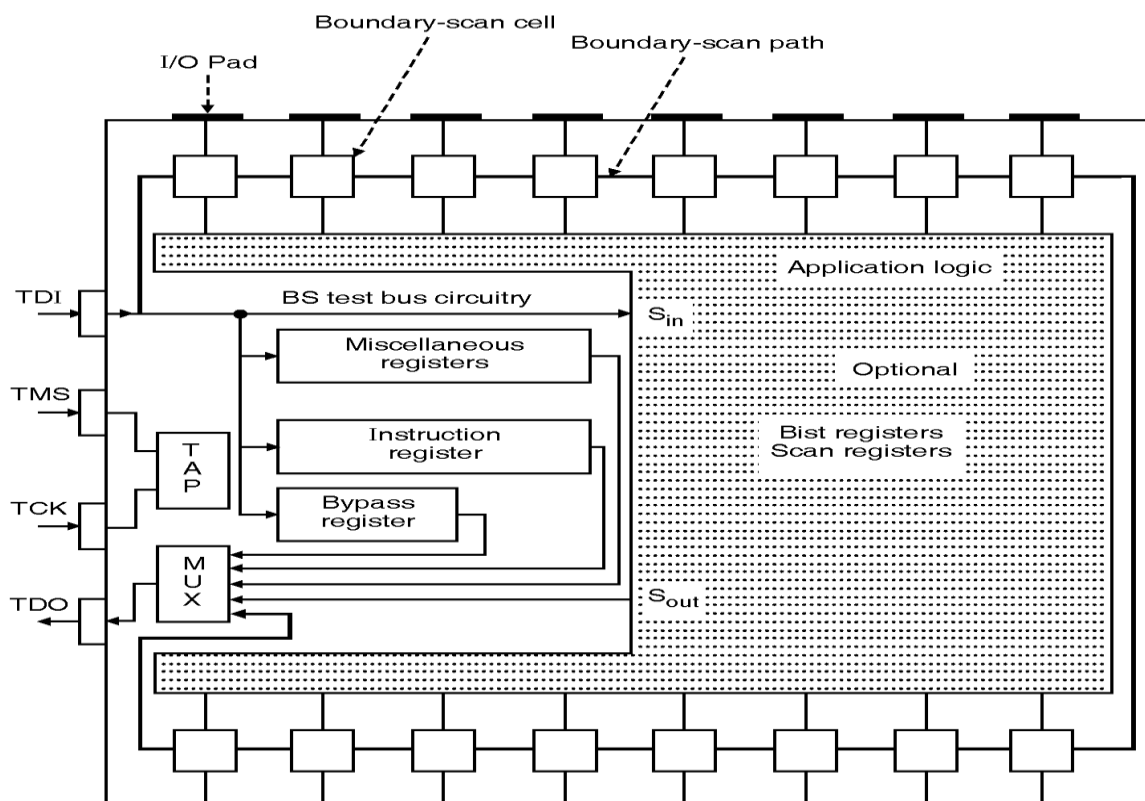
Functional testing checks the circuit's functionality, while parametric testing checks the circuit's electrical characteristics. Defect testing checks for any defects or faults in the circuit.

2. **Package testing:** After the dies are packaged, they undergo package testing. Package testing checks the packaged devices for any defects or faults that may have occurred during the packaging process. The tests performed during package testing include visual inspection, electrical testing, and thermal testing.
3. **Final testing:** Final testing is the last stage of testing in CMOS technology. It involves testing the final product after assembly. The tests performed during final testing include functional testing, reliability testing, and environmental testing. Functional testing checks the product's functionality, reliability testing checks the product's reliability, and environmental testing checks the product's performance under different environmental conditions.

The specific tests performed during each stage of testing may vary depending on the product and its application. The test procedures are designed to ensure that the product meets the required specifications, quality standards, and customer expectations. The testing process is critical to ensuring the reliable operation of CMOS products and delivering high-quality products to customers.

### 3. Boundary Scan in CMOS

Boundary scan is a test technique using scan methodology, involving digital services, digital devices, designed with scan flip flops placed between each device pin and the internal logic. These registers can control and observe signal values present at each input and output pin and are connected together in serial fashion to form a data register chain, called boundary scan shift register with shift and update stages.



The update stage latch prevents output from rippling as data is shifted through the shift register during scan operation. Figure above shows how the boundary scan registers can be connected

in an ASIC. Test sets generated by automatic test pattern generation can be scanned into these boundary scan registers through scan in port such that test stimuli are applied parallel, circuit response can be captured in parallel by boundary scan registers connected between internal logic and output pins and scanned out through scan out port.

### **Boundary Scan Standards :**

To better address problems of board-level testing, several design for testability standards have been developed. The primary goal of these proposed standards is to ensure that chips of VLSI complexity contain a common denominator of DFT circuitry that will make the test development and testing of boards containing these chips significantly more effective and less costly. Some of these initiatives are known as the Joint Test Action Group (JTAG).

### **Advantages of Boundary Scan :**

No need for complex testers in PCB testing.

The test engineer's work is simplified and efficient.

The time spent on test pattern generation and application is reduced.

Fault coverage is increased.

### **Boundary Scan 1149.1 Standard :**

The standards discussed above deal primarily with the use of a test bus which will reside on a board, the protocol associated with this bus, elements of a bus master which controls the bus, I/O ports that tie a chip to the bus, and some control logic that must reside on a chip to interface the test bus ports to the DFT hardware residing on the application portion of the chip. In addition, the JTAG boundary scan and IEEE 1149.1 standards also require that a boundary-scan register exist on the chip. The description of a board-level test bus presented on IEEE 1149.1. Figure below shows a general form of a chip which supports IEEE 1149.1. The application logic represents the normal chip design prior to the inclusion of logic required to support IEEE 1149.1. The normal I/O terminals of the application logic are connected through boundary-scan cells to the chips I/O pads. The test-bus circuitry, also referred to as the bus slave consists of the boundary-scan registers, a 1-bit bypass register, an instruction register, several miscellaneous registers, and the TAP. The boundary-scan bus consists of four lines, that are a test clock (TCK), a test mode signal (TMS), the TDI line, and the TDO line. Test instructions and test data are sent to a chip over the TDI line. Test results and status information are sent from a chip over the TDO line serially. The sequence of operations are controlled by a bus master, which can be either ATE or a component that interfaces to a higher-level test bus. Control of the test-bus circuitry is primarily carried out by the TAP, which responds to the state transitions on the TMS line.

## **4. Built-in Self Test (BIST) :**

As the complexity of VLSI circuits and as overall system complexity increases, test generation and application becomes an expensive and not always very effective means of testing. Further there are also very difficult problems associated with the high speeds at which many VLSI systems are designed to operate. Such problems require the use of very sophisticated, but not always affordable, test equipments. Built-in Self Test (BIST) is another solution. Figure below shows the Built-in Self Test system



The AND gate shown in figure is fault-modeled for inputs and the outputs. This results in  $n + 2$  tests for an  $n$ -input AND gate. The input tested is controlling input which determines the value appears on the output. Input pattern of all 1s tests for the output. It is not necessary to test for an output fault because any input test also detect the output. Further, output is detected without detecting any input fault if two or more inputs are at logic low.

OR Gate Fault Model:

An  $n$ -input OR gate, requires  $n + 2$  tests. Further, the input values are the complement of the values for an AND gate. The input tested is set to 1 and all other inputs are set to 0.

Inverter Fault Model:

The Inverter is modeled with output. When failed to invert a transistor and both stuck-at faults are detected.

\*\*\*\*\*

The chip manufacturing process is prone to defects and the defects are commonly referred as faults. A fault is testable if there exists a well-specified procedure to expose it in the actual silicon. To make the task of detecting as many faults as possible in a design, we need to add additional logic; Design for testability (DFT) refers to those design techniques that make the task of testing feasible. In this article we will be discussing about the most common DFT technique for logic test, called Scan and ATPG. Before going into Scan and ATPG basics, let us first understand the concept of fault model.

## 6. Fault Models

Fault models abstract the behavior of manufacturing defects so that test vectors can be generated to detect them.

- Functional Defects : Stuck-at Fault Model
- Current defects : Pseudo Stuck-at Fault Model ( $I_{DDQ}$ )
- Speed defects: At-speed Fault Model, Path Delay Fault Model

However in this article we will be discussing about two most common fault models: stuck-at and at-speed fault models.

### 1. Stuck-at Faults

This is the most common fault model used in industry. It models manufacturing defects which occurs when a circuit node is shorted to VDD (stuck-at-1 fault) or GND (stuck-at-0 fault) permanently. The fault can be at the input or output of a gate. Thus a simple 2-input AND gate has six possible stuck-at faults.

In the circuit shown in Figure 1, suppose we have a stuck-at-0 fault at the output of an AND gate. Note one important thing, there are three input ports in the circuit, thus we can have a combination of eight different inputs or patterns {000, 001, 010, 011, 100, 101, 110, 111}; out of the eight patterns, only two patterns {011, 111} will be able to detect this fault because with

rest of the patterns the expected output will be same as the actual circuit output in the presence of this s-a-0 fault. This is a small circuit so we can easily find the pattern that can detect this fault, but what about much bigger circuits? Well we don't have to worry about it as the CAD tools (ATPG tools) will do that for us. The ATPG tools will try to generate the stuck-at fault patterns required to test all the possible fault locations using complex algorithms, but if it is unable to find patterns for few faults, then it will classify those faults as untestable.

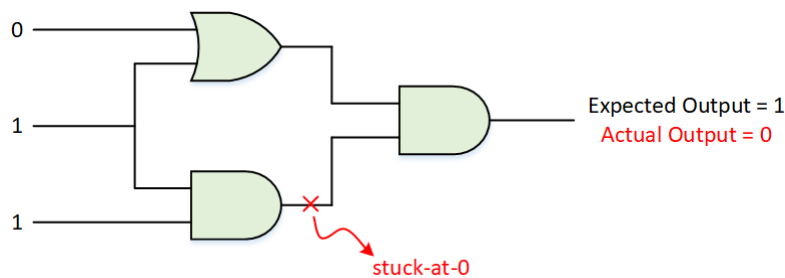


Figure 1: stuck-at-0 fault in a circuit

## 2. At-speed Faults

It models the manufacturing defects that behave as gross delays on gate input-output ports. So each port is tested for logic 0-to-1 transition delay (slow-to-rise fault) or logic 1-to-0 transition delay (slow-to-fall fault). Like stuck-at faults, the at-speed fault can be at the input or output of a gate, thus a simple 2-input AND gate has six possible at-speed faults.

In the circuit shown in Figure 2, suppose we have a slow-to-fall fault at the output of an AND gate. As shown, a slower 1-to-0 transition at the output of AND gate can affect the value captured by the Flop 2 at its capture edge. It is important to note that only with an initial state '1' in Flop 1 and 010 at the input, we will be able to detect this fault. And like stuck-at fault pattern generation, the ATPG tools will try to generate the at-speed fault patterns required to test all the possible fault locations.

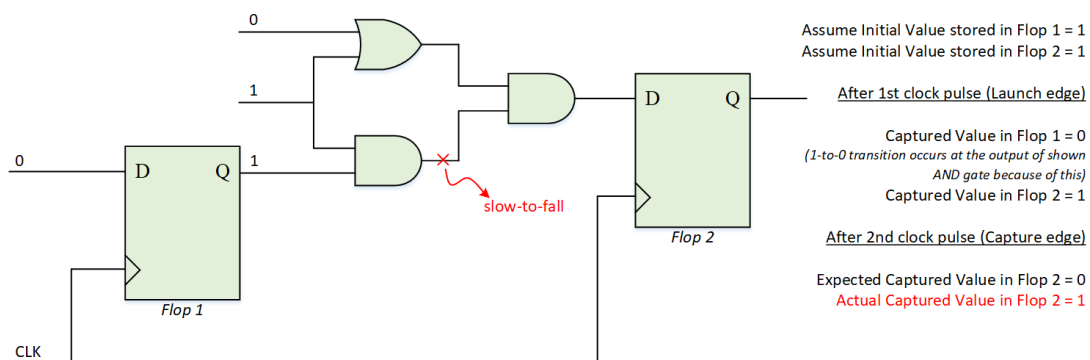


Figure 2: slow-to-fall fault in a circuit

## Scan and ATPG

Scan is the internal modification of the design's circuitry to increase its test-ability. ATPG stands for Automatic Test Pattern Generation; as the name suggests, this is basically the generation of test patterns. In other words, we can say that Scan makes the process of pattern generation easier for detection of the faults we discussed earlier.

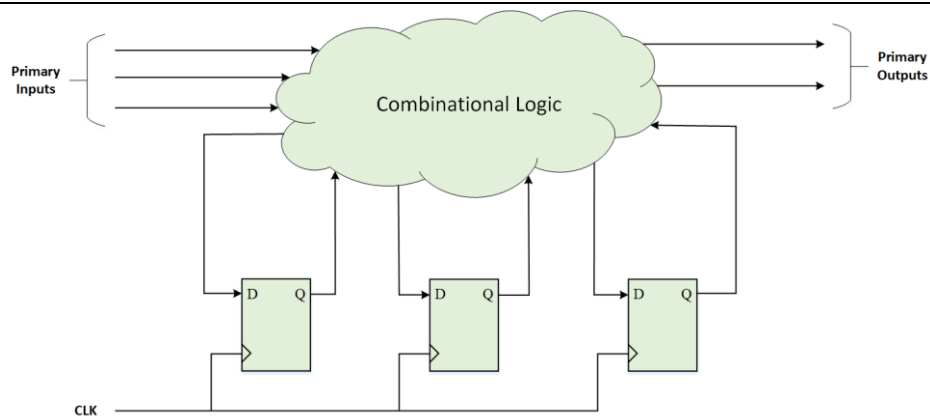


Figure 3: A typical sequential circuit (before scan insertion)

To test a fault we need to initialize the flops to the required values as we had shown while discussing about stuck-at faults and at-speed faults. In a bigger sequential circuit (without scan), it is difficult to control the flop's value through primary inputs and observe the captured response in primary outputs. To solve this issue we do 'Scan Insertion' during synthesis.

The goal of 'Scan Insertion' is to make a difficult-to-test sequential circuit behave (during testing process) like an easier-to-test combinational circuit. Achieving this goal involves two steps –

### 1. Converting Regular Flop to Scan Flop

All the flops in the design are converted into scan flops (as shown in Figure 4), except –

- The ones that are excluded by user. These are called non-scan flops.
- The ones that have DFT DRC violation(s).

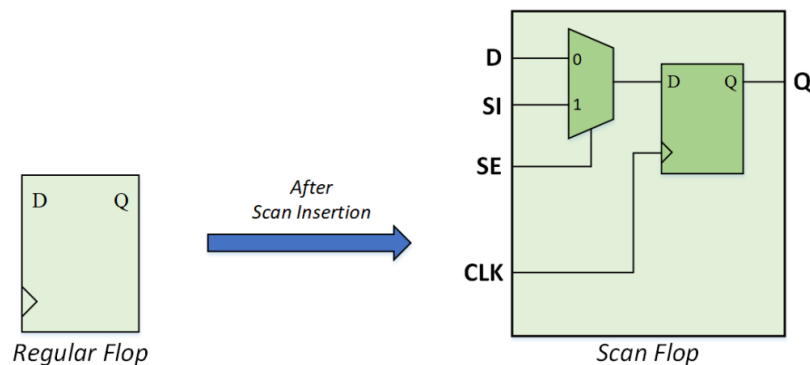


Figure 4: Regular flop vs Scan flop

### 2. Stitching the Scan Flops to form Scan Chains

The scan flops are stitched to form scan chain(s) (as shown in Figure 5). The number of scan chains depends upon various user inputs like –

- Length of scan chain
- Clock domain mixing
- Power domain mixing
- Voltage domain mixing



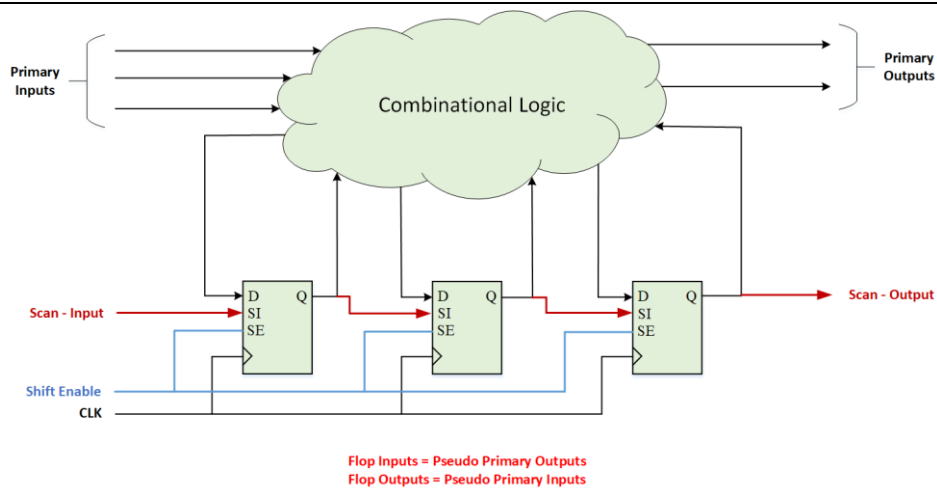


Figure 5: A typical sequential circuit compatible for Scan and ATPG (after scan insertion)

To initialize any flop to a value (refer the Figure 5), we simply make the SE = 1, such that SI to Q path is activated and we shift in the required values serially through a top level primary input called Scan-Input. Once the required values are loaded to the flops, we capture the values from combinational circuit by making SE = 0. And to observe the captured response we make the SE = 1 and serially shift out the captured data through a primary output called Scan-Output. Thus in a way, we can say the scan flop's output (Q) act as pseudo primary output of the design and the scan flop's input (D) act as pseudo primary inputs to the design, thereby making it a pseudo combination circuit.

Once the patterns are generated, the expected response of the circuit for each pattern is obtained in pre-silicon. The expected responses along with the patterns are then stored in the memory of Automatic Test Equipment (ATE). In post-silicon, the manufactured chip is tested using the ATE, which loads the pattern and compares it with the expected response for pass or fail status.

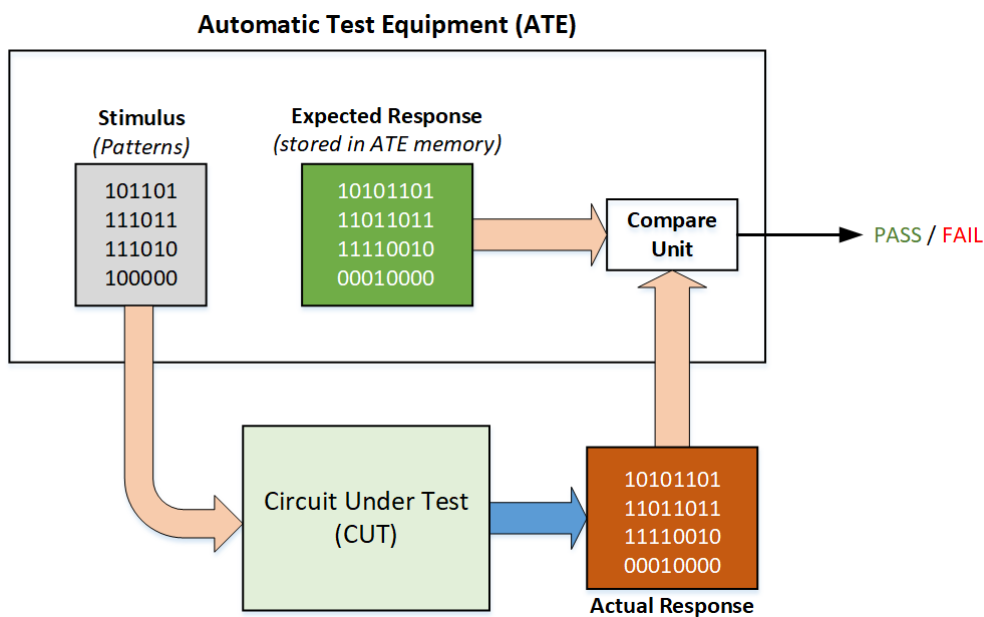


Figure 6: A schematic showing how testing works

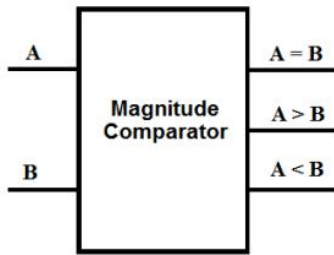
7. **Fault simulation** is a technique used to simulate the occurrence of faults or defects in digital circuits to evaluate the effectiveness of the design for testability (DFT) techniques and test patterns. The purpose of fault simulation is to identify and analyze the circuit's response to different fault conditions and assess the test coverage, which is the percentage of faults that can be detected by the test patterns.

Here are the key steps involved in the fault simulation process:

1. **Fault model creation:** The first step in fault simulation is to create a fault model, which defines the types of faults that can occur in the circuit. Common fault models used in fault simulation include stuck-at faults, bridging faults, and delay faults.
2. **Test pattern generation:** The next step is to generate test patterns that can be used to detect the faults in the circuit. Test patterns are generated using test generation tools, and the effectiveness of the test patterns is evaluated by simulating the circuit with the fault models.
3. **Fault simulation:** In the fault simulation process, the test patterns are applied to the circuit with the fault models, and the circuit's response is analyzed. The analysis involves comparing the expected output of the circuit with the actual output under fault conditions.
4. **Test coverage analysis:** The test coverage analysis involves evaluating the percentage of faults that can be detected by the test patterns. The goal is to achieve high test coverage, which indicates that a high percentage of faults can be detected by the test patterns.
5. **Fault diagnosis:** Fault diagnosis involves identifying the location and type of faults that are not detected by the test patterns. Fault diagnosis is an essential step in fault simulation because it helps to identify design or manufacturing issues that need to be addressed to improve the test coverage and overall product quality.

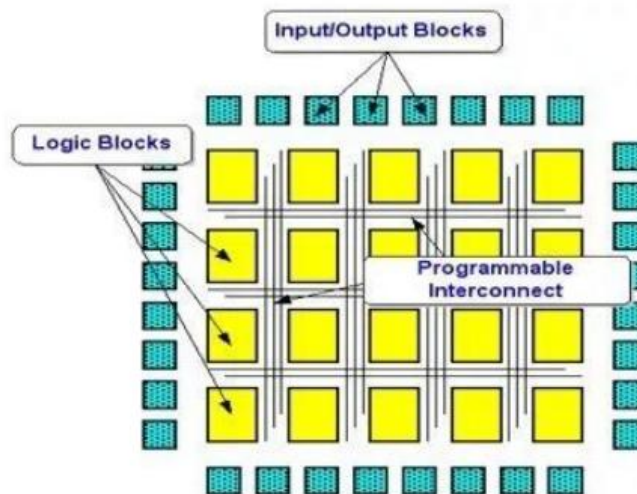
Overall, fault simulation is an important tool in the DFT process, as it helps to evaluate the effectiveness of the test patterns and identify any design or manufacturing issues that may affect the circuit's performance and reliability. By simulating faults in the circuit and analyzing the test coverage, designers can improve the circuit's testability, reduce the cost of testing, and ensure high-quality products.

8. **Magnitude Comparator** A magnitude comparator determines the larger of two binary numbers. To compare two unsigned numbers A and B, compute  $B - A = B + A + 1$ . If there is a carry-out,  $A \leq B$ ; otherwise,  $A > B$ . A zero detector indicates that the numbers are equal.



Inputs				Outputs		
A <sub>1</sub>	A <sub>0</sub>	B <sub>1</sub>	B <sub>0</sub>	A > B	A = B	A < B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

9. **Field Programmable Gate Array (FPGA)** Fully fabricated FPGA chips containing thousands or even more, of logic gates with programmable interconnects, are available to users for their custom hardware programming to realize desired functionality. This design style provides a means for fast prototyping and also for cost-effective chip design, especially for low-volume applications. A typical field programmable gate array (FPGA) chip consists of I/O buffers, an array of configurable logic blocks (CLBs), and programmable interconnect structures. The programming of the interconnects is accomplished by programming of RAM cells whose output terminals are connected to the gates of MOS pass transistors. Thus, the signal routing between the CLBs and the I/O blocks is accomplished by setting the configurable switch matrices accordingly. The general architecture of an FPGA chip from Xilinx . showing the locations of switch matrices used for interconnect routing.



**Important points to know about Field-Programmable Gate Arrays (FPGAs):**

1. FPGAs are reprogrammable integrated circuits that can be programmed and reprogrammed to perform a wide variety of digital logic functions.
2. FPGAs are made up of programmable logic blocks and programmable interconnects, which can be configured and connected to create custom logic pathways between the blocks.
3. FPGAs typically have other programmable features such as memory, input/output interfaces, and specialized digital signal processing (DSP) blocks.
4. FPGAs are highly versatile and adaptable to a wide range of applications, including digital signal processing, high-performance computing, and embedded systems.

5. FPGAs are commonly used in prototyping and testing of new digital circuits, as they allow for quick and easy implementation of custom logic functions.
6. FPGAs can be more expensive than other types of integrated circuits, and their programmable nature can make them more complex to design and debug.
7. FPGAs may not always be the most efficient solution for certain applications, as they may require more power or take up more space than other integrated circuits.
8. FPGAs are commonly programmed using hardware description languages (HDLs) such as VHDL or Verilog.
9. FPGAs can be programmed using either a hardware programming language, which allows for greater control and customization, or a high-level language, which is easier to use but may have more limited functionality.
10. FPGAs can offer several advantages over application-specific integrated circuits (ASICs), including faster time-to-market, greater design flexibility, and lower development costs.

### 10. Test Pattern Generation (TPG) and Automatic Test Pattern Generation (ATPG)

Feature	Test Pattern Generation (TPG)	Automatic Test Pattern Generation (ATPG)
<b>Definition</b>	A technique used to manually generate test patterns for a circuit.	An automated process that uses algorithms to generate test patterns.
<b>Input required</b>	Circuit design and knowledge of its expected behavior.	Circuit design, fault models, and timing constraints.
<b>Methodology</b>	Manually creating test patterns based on engineering expertise.	Using algorithms to generate test patterns that achieve complete fault coverage.
<b>Scope</b>	Used for simpler circuits with known faults.	Used for complex circuits with unknown faults.
<b>Efficiency</b>	Time-consuming and may not achieve complete fault coverage.	More efficient and can achieve complete fault coverage.
<b>Accuracy</b>	Highly dependent on the expertise of the engineer creating the test patterns.	More accurate and can detect hard-to-find faults.
<b>Cost</b>	Relatively low cost, as it only requires human labor.	Higher cost, as it requires expensive software and computing resources.
<b>Applications</b>	Useful for simple circuits with known faults, or for creating a limited set of test patterns.	Essential for complex circuits with unknown faults, or for creating large numbers of test patterns.

### 11. Adhoc Testing

#### What is Adhoc Testing?

When a software testing performed without proper planning and documentation, it is said to be Adhoc Testing. Such kind of tests are executed only once unless we uncover the defects.

Adhoc Tests are done after formal testing is performed on the application. Adhoc methods are the least formal type of testing as it is NOT a structured approach. Hence, defects found using this method are hard to replicate as there are no test cases aligned for those scenarios.

Testing is carried out with the knowledge of the tester about the application and the tester tests randomly without following the specifications/requirements. Hence the success of Adhoc testing depends upon the capability of the tester, who carries out the test. The tester has to find defects without any proper planning and documentation, solely based on tester's intuition.

### **When to Execute Adhoc Testing ?**

Adhoc testing can be performed when there is limited time to do exhaustive testing and usually performed after the formal test execution. Adhoc testing will be effective only if the tester has in-depth understanding about the System Under Test.

### **Forms of Adhoc Testing :**

1. **Buddy Testing:** Two buddies, one from development team and one from test team mutually work on identifying defects in the same module. Buddy testing helps the testers develop better test cases while development team can also make design changes early. This kind of testing happens usually after completing the unit testing.
2. **Pair Testing:** Two testers are assigned the same modules and they share ideas and work on the same systems to find defects. One tester executes the tests while another tester records the notes on their findings.
3. **Monkey Testing:** Testing is performed randomly without any test cases in order to break the system.

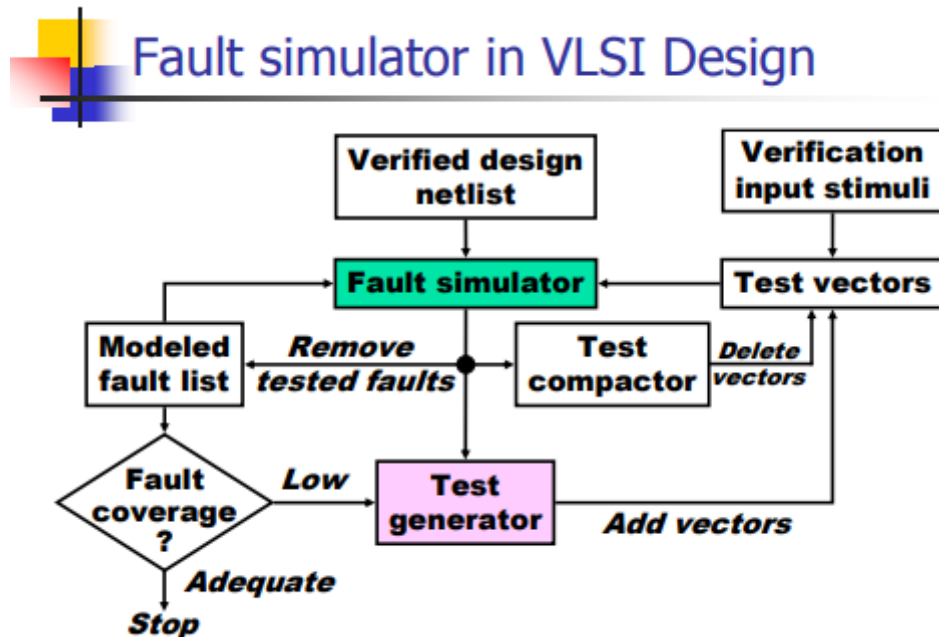
### **Various ways to make Adhoc Testing More Effective**

1. **Preparation:** By getting the defect details of a similar application, the probability of finding defects in the application is more.
2. **Creating a Rough Idea:** By creating a rough idea in place the tester will have a focussed approach. It is NOT required to document a detailed plan as what to test and how to test.
3. **Divide and Rule:** By testing the application part by part, we will have a better focus and better understanding of the problems if any.
4. **Targeting Critical Functionalities:** A tester should target those areas that are NOT covered while designing test cases.
5. **Using Tools:** Defects can also be brought to the lime light by using profilers, debuggers and even task monitors. Hence being proficient in using these tools one can uncover several defects.
6. **Documenting the findings:** Though testing is performed randomly, it is better to document the tests if time permits and note down the deviations if any. If defects are found, corresponding test cases are created so that it helps the testers to retest the scenario.

## 12. Fault Simulation

Simulation serves two distinct purposes in electronic design. First, it is used to verify the correctness of the design, and second, it verifies the tests. Fault simulation can be classified as

1. Serial Fault Simulation
2. Parallel Fault Simulation
3. Concurrent Fault Simulation
4. Nondeterministic Fault Simulation.



1. **Fault models:** Fault models define the types of faults that are injected into the circuit for simulation, such as stuck-at-0, stuck-at-1, or bridging faults. It's important to select the appropriate fault model to evaluate the circuit's fault tolerance.
2. **Test vectors:** Test vectors are input patterns that are applied to the circuit during simulation. It's important to use a comprehensive set of test vectors that can detect as many faults as possible to achieve high fault coverage.
3. **Coverage metrics:** Coverage metrics measure the effectiveness of the test vectors in detecting faults. Examples of coverage metrics include stuck-at fault coverage, transition fault coverage, and path delay fault coverage. High coverage for all relevant fault models is important to ensure the circuit's reliability.
4. **Simulation speed:** Fault simulation can be computationally intensive, so it's important to choose a fast and efficient fault simulator to minimize simulation time and improve design turnaround time.
5. **Analysis and debugging:** After the simulation is complete, it's important to analyze the results and identify any undetected faults or error-prone areas in the design. Debugging techniques such as fault isolation and diagnosis can be used to pinpoint the root cause of faults and improve the circuit's reliability.
6. **Verification:** Fault simulation is just one part of the overall verification process for digital circuits. It's important to integrate fault simulation with other verification techniques such as timing analysis, functional simulation, and formal verification to achieve a high level of confidence in the circuit's functionality and reliability.

## 1. Verilog Basics:

### Basics of Verilog

Verilog is a hardware description language (HDL) used to design and describe digital circuits. Verilog programs are composed of modules, which are the basic building blocks of the design. Modules contain input and output ports, which allow them to interact with other modules or the outside world.

#### Example:

```
module my_module(input A, input B, output C);  
  
    assign C = A & B;  
  
endmodule
```

In the example above, the my\_module module has two input ports (A and B) and one output port (C). The assign statement assigns the output C to the bitwise AND of the inputs A and B.

### Operators

Verilog supports a variety of operators for performing arithmetic, bitwise, and logical operations. Here are some examples:

#### Arithmetic Operators:

- *+ Addition*
- *- Subtraction*
- *\* Multiplication*
- */ Division*
- *% Modulus*

#### Example:

```
reg a = 4;  
reg b = 2;  
reg c = a + b; // c will be 6
```

#### Bitwise Operators:

- *& Bitwise AND*
- *| Bitwise OR*
- *^ Bitwise XOR*
- *~ Bitwise NOT*
- *<< Bitwise left shift*
- *>> Bitwise right shift*

### Example:

```
reg a = 2'b10;
reg b = 2'b11;
reg c = a & b; // c will be 2'b10
```

### Logical Operators:

- *&&* Logical AND
- *//* Logical OR
- *!* Logical NOT

### Example:

```
reg a = 1'b1;
reg b = 1'b0;
reg c = !(a && b); // c will be 1'b1
```

### Data Types

Verilog supports several data types for representing different types of data, such as integers, booleans, and arrays. Some of the commonly used data types include:

#### Integer Types:

- *bit* A single bit
- *reg* A one-dimensional array of bits
- *wire* A one-dimensional array of bits that can be used for interconnecting modules

### Example:

```
reg [7:0] a = 8'b10101010;
wire [7:0] b;
my_module m(a, b);
```

In the example above, a is an 8-bit register initialized to the binary value 10101010. b is a wire that is used to connect the my\_module module to the outside world.

#### Boolean Types:

- *logic* A one-dimensional array of bits that can represent Boolean values

### Example:

```
logic [3:0] a = 4'b1100;
```

In the example above, a is a 4-bit logic array initialized to the binary value 1100.

#### Arrays:

- *int* An integer type that can represent signed or unsigned values



- **real** A floating-point type for representing real numbers
- **parameter** A constant value that can be used to parameterize modules

**Example:**

```
parameter WIDTH = 8;
reg [WIDTH-1:0] a = WIDTH'b10101010;
```

In the example above, **WIDTH** is a parameter that specifies the width of the a register. The **WIDTH-1:0** notation is used to specify the range of the register.

#### 4. Continuous Assignments:

- Continuous assignments are used to assign a value to a signal continuously.
- Continuous assignments use the assign keyword and the = operator to assign a value to a wire.
- Continuous assignments are used to implement combinational logic in a Verilog module.

##### Continuous Assignments

Continuous assignments in Verilog allow you to assign values to nets (wires) in a module continuously, without the need for a procedural block. This means that the assignment is made continuously as the input values change. Continuous assignments are always executed in the same order, from left to right.

Continuous assignments are defined using the **assign** keyword in Verilog. The syntax for a continuous assignment is as follows:

```
assign net = expression;
```

Here, **net** is the name of the wire to be assigned, and **expression** is the value to be assigned to the wire.

Example:

```
module and_gate(input a, input b, output y);
    assign y = a & b;
endmodule
```

In the example above, a continuous assignment is used to assign the value of the bitwise AND of **a** and **b** to the output **y** of the **and\_gate** module. Whenever the inputs **a** or **b** change, the output **y** will be updated automatically.

Continuous assignments can also be used to assign the output of a function to a wire. Here's an example:

```

module my_module(input a, input b, output y);

    function [1:0] my_function;
        input [1:0] a;
        input [1:0] b;
        begin
            my_function = a + b;
        end
    endfunction

    assign y = my_function(a, b);

endmodule

```

In the example above, a continuous assignment is used to assign the output of the **my\_function** function to the output **y** of the **my\_module** module. Whenever the inputs **a** or **b** change, the output **y** will be updated automatically.

Continuous assignments can also be used to assign values to an array of wires. Here's an example:

```

module my_module(input a, input b, output [3:0] y);

    assign y = {a, b, a & b, a | b};

endmodule

```

In the example above, a continuous assignment is used to assign values to the 4-bit output array **y** of the **my\_module** module. The assignment uses concatenation to assign the values of **a**, **b**, **a & b**, and **a | b** to the array elements **y[3]**, **y[2]**, **y[1]**, and **y[0]**, respectively. Whenever the inputs **a** or **b** change, the output **y** will be updated automatically.

## 5. Sequential and Parallel Statement Groups:

- Sequential statements are executed one after the other in the order they are written in the code.
- Examples of sequential statements include if-else statements, case statements, and loops.
- Parallel statement groups execute concurrently with other statements in the module.
- Examples of parallel statement groups include always blocks, initial blocks, and concurrent assignments.
- Always blocks are used to define the behavior of registers and flip-flops in a circuit.
- Initial blocks are used to initialize the values of signals and variables at the start of simulation.

## Sequential and Parallel Statement Groups

Sequential and parallel statement groups in Verilog allow you to group together multiple statements and specify their order of execution. There are two types of statement groups:

### 1. Sequential Statement Group

A sequential statement group consists of a block of statements that are executed one after the other in the specified order. The statements in a sequential statement group are enclosed within a **begin** and **end** block.

Example:

```
module seq_example(A, B, C, D);
  input A, B;
  output C, D;

  reg [3:0] data;

  always @(A, B) begin
    data[0] = A;
    data[1] = B;
    data[2] = A & B;
    data[3] = A | B;

    begin
      C = data[2];
      D = data[3];
    end
  end

end
endmodule
```

In the example above, the **seq\_example** module uses an **always** block with a sequential statement group to assign the output signals **C** and **D** to the values of the third and fourth bits of the **data** register. The **begin** and **end** blocks enclose the assignment statements and ensure that they are executed sequentially.

### 2. Parallel Statement Group

A parallel statement group consists of a block of statements that are executed simultaneously and in no specified order. The statements in a parallel statement group are enclosed within a pair of curly braces **{}**.

Example:

```
module par_example(A, B, C, D);
  input A, B;
  output C, D;

  reg [3:0] data;

  always @(A, B) begin
    data[0] = A;
    data[1] = B;
    data[2] = A & B;
    data[3] = A | B;

    {C, D} = {data[2], data[3]};

  end
endmodule
```

In the example above, the **par\_example** module uses an **always** block with a parallel statement group to assign the output signals **C** and **D** to the values of the third and fourth bits of the **data** register. The **{}** curly braces enclose the assignment statements and ensure that they are executed in parallel and in no specified order.

Sequential and parallel statement groups are useful for organizing and controlling the execution order of statements in a Verilog module. Sequential statement groups ensure that the statements are executed in the specified order, while parallel statement groups allow multiple statements to be executed simultaneously and in no specified order.

## 6. Timing Control:

- Timing control is used to specify when a particular section of code should be executed.
- There are two types of timing control: level-sensitive and edge-sensitive.
- Level-sensitive timing control is used to specify when a section of code should be executed based on the value of a signal.
- Edge-sensitive timing control is used to specify when a section of code should be executed based on the change in value of a signal.

## 7. Delays:

- Delays are used to specify the amount of time that should pass before a particular section of code is executed.
- Delays can be specified using the **#** operator, followed by the delay time in units of time.

- Delays can also be specified using the \$delay system task.

## Delay

Delay is the time taken by a signal to propagate through a combinational logic circuit. It is an important parameter in timing analysis and can be specified in Verilog using delay expressions.

There are two types of delays in Verilog:

### 1. Constant Delay

A constant delay is a fixed time delay specified as a number followed by a time unit, such as 1ns, 10ps, etc. It represents the delay through a combinational logic gate.

Example:

```
module delay_example(A, B, C);  
  input A, B;  
  output C;  
  
  assign #1 C = A & B;  
endmodule
```

In the example above, the delay\_example module uses a bitwise AND operator to compute the output C from the inputs A and B. The #1 delay expression specifies a constant delay of 1 time unit.

### 2. Net Delay

A net delay is a delay that represents the time taken by a signal to propagate through a wire or a net. It is specified as a number followed by a time unit, and is added to the delay of the gate that drives the net.

Example:

```
module delay_example(A, B, C);  
  input A, B;  
  output C;  
  
  wire D;  
  assign D = A & B;  
  assign #1 C = D;  
endmodule
```

In the example above, the delay\_example module uses a wire D to connect the inputs A and B to a bitwise AND operator. The #1 delay expression specifies a delay of 1 time unit for the output C, which includes the delay of the AND gate and the delay of the wire D.

Delay expressions can be used to model timing constraints in the design and ensure that the circuit meets timing requirements.

It is important to note that delays in Verilog are not always deterministic and can vary depending on the implementation of the circuit and the timing characteristics of the hardware. Therefore, delay should be used with caution and verified using timing analysis tools.

**Example:**

### 8. Tasks and Functions:

- Tasks and functions are used to encapsulate a section of code that can be reused in different parts of the design.
- Tasks are used for executing a sequence of statements and do not return a value.
- **Functions** are used for executing a sequence of statements and return a value.
- Tasks and functions can have input and output arguments.

### Tasks

A task is a reusable block of code that can contain multiple procedural statements. It can be called from other parts of the Verilog code using a task call statement.

**Example:**

```
task add(input [7:0] A, B, output [7:0] C);
    C = A + B;
endtask

module testbench;
    reg [7:0] A, B;
    wire [7:0] C;

    initial begin
        A = 8'h23;
        B = 8'h45;
        add(A, B, C);
        $display("C = %h", C);
    end
endmodule
```

In the example above, the **add** task takes two 8-bit input arguments **A** and **B** and an output argument **C**. It adds **A** and **B** and assigns the result to **C**. The **testbench** module calls the **add** task and displays the value of **C**.

Tasks can be useful for creating reusable blocks of code and improving the readability of the Verilog code.

### Functions

A function is a reusable block of code that returns a value. It can be called from other parts of the Verilog code using a function call statement.

**Example:**

```
function [7:0] add(input [7:0] A, B);
    add = A + B;
endfunction

module testbench;
    reg [7:0] A, B;
    wire [7:0] C;

    initial begin
        A = 8'h23;
        B = 8'h45;
        C = add(A, B);
        $display("C = %h", C);
    end
endmodule
```

In the example above, the **add** function takes two 8-bit input arguments **A** and **B**. It adds **A** and **B** and returns the result. The **testbench** module calls the **add** function and assigns the result to **C**.

Functions can be useful for creating reusable blocks of code and improving the readability of the Verilog code. They can also be used to compute intermediate values that are used in multiple places in the code.

## 9. Control Statements

Control statements are used in programming languages to alter the flow of program execution. There are three types of control statements:

1. Sequential: The default flow of the program is sequential. Statements are executed one after another in the order they are written in the program.
2. Selection: Control statements such as if-else and switch are used for selection, where a block of code is executed depending on the condition that is met.
3. Iteration: Control statements such as for, while, and do-while are used for iteration, where a block of code is executed repeatedly until a condition is met.

## 10. Blocking and Non-Blocking Assignments

In Verilog, there are two types of assignments: blocking and non-blocking. Blocking assignments use the "=" operator, and non-blocking assignments use the "<=" operator.

### Blocking Assignments

In a blocking assignment, the statement on the right-hand side of the "=" operator is evaluated immediately, and the value is assigned to the variable on the left-hand side. The next statement in the code is not executed until the current statement has completed.

**Example:**

```
always @(posedge clk)
begin
    a = b;
    c = a;
end
```

In the example above, the assignment **a = b** is a blocking assignment. The value of **b** is evaluated immediately, and the value is assigned to **a**. The next statement, **c = a**, is not executed until the blocking assignment is complete.

### **Non-Blocking Assignments**

In a non-blocking assignment, the statement on the right-hand side of the "<=" operator is evaluated at the end of the current time step, and the value is assigned to the variable on the left-hand side at the beginning of the next time step. This means that all non-blocking assignments in a given always block are executed simultaneously.

**Example:**

```
always @(posedge clk)
begin
    a <= b;
    c <= a;
end
```

In the example above, the assignment **a <= b** is a non-blocking assignment. The value of **b** is not assigned to **a** until the beginning of the next time step. The assignment **c <= a** is also a non-blocking assignment, and it will be executed simultaneously with the first non-blocking assignment.

## **11. If-Else Statements**

The if-else statement is a selection control statement that allows the programmer to execute a block of code if a condition is met, and a different block of code if the condition is not met.

**Example:**

```
if (condition)
begin
    // Block of code to execute if the condition is true
end
else
begin
    // Block of code to execute if the condition is false
end
```

In the example above, the condition is evaluated, and if it is true, the block of code in the first "begin-end" block is executed. If the condition is false, the block of code in the second "begin-end" block is executed.



## 12. Case Statements

The case statement is a selection control statement that allows the programmer to execute a block of code based on the value of a variable.

Example:

```
case (variable)
  value1: begin
    // Block of code to execute if variable == value1
  end
  value2: begin
    // Block of code to execute if variable == value2
  end
  default: begin
    // Block of code to execute if variable does not match any of the previous
  end
endcase
```

In the example above, the value of the variable is evaluated,

## 13. For Loop

The for loop is an iteration control statement that allows the programmer to execute a block of code a specified number of times. It has three parts: the initialization, the condition, and the increment.

Example:

```
for (i = 0; i < n; i = i + 1)
begin
  // Block of code to execute n times
end
```

In the example above, the for loop initializes *i* to 0, checks if *i* is less than *n*, and increments *i* by 1 after each iteration of the block of code.

## 14. While Loop

The while loop is an iteration control statement that allows the programmer to execute a block of code repeatedly while a condition is true.

Example:

```
while (condition)
begin
  // Block of code to execute while the condition is true
end
```

In the example above, the while loop checks the condition before each iteration of the block of code. If the condition is true, the block of code is executed. If the condition is false, the loop exits.

## 15. Repeat Loop

The repeat loop is an iteration control statement that allows the programmer to execute a block of code a specified number of times. It has one part: the count.

**Example:**

```
repeat (n)
begin
  // Block of code to execute n times
end
```

In the example above, the repeat loop executes the block of code **n** times.

## 16. Forever Loop

The forever loop is an iteration control statement that allows the programmer to execute a block of code repeatedly without any condition or iteration count.

**Example:**

```
forever
begin
  // Block of code to execute repeatedly
end
```

In the example above, the forever loop executes the block of code repeatedly without any condition or iteration count.

## 17. Rise, Fall, Min, and Max Delays

Delays are used in Verilog to specify timing relationships between signals. There are four types of delays: rise delay, fall delay, minimum delay, and maximum delay.

### Rise Delay

The rise delay is the time it takes for a signal to transition from 10% to 90% of its final value.

**Example:**

```
assign y = #5 x;
```

In the example above, the rise delay is 5 time units. The signal **y** will transition from 10% to 90% of its final value 5 time units after the signal **x** makes a transition.

### Fall Delay

The fall delay is the time it takes for a signal to transition from 90% to 10% of its final value.

**Example:**

```
assign y = #(2,3) x;
```

In the example above, the fall delay is 3 time units. The signal **y** will transition from 90% to 10% of its final value 3 time units after the signal **x** makes a transition.

### Minimum Delay

The minimum delay is the shortest amount of time it takes for a signal to propagate through a logic gate.

Example:

```
assign y = #0.5 x;
```

In the example above, the minimum delay is 0.5 time units. The signal **y** will be updated 0.5 time units after the signal **x** is updated.

### Maximum Delay

The maximum delay is the longest amount of time it takes for a signal to propagate through a logic gate.

Example:

```
assign y = #2 x;
```

In the example above, the maximum delay is 2 time

## 18. Behavioural Coding Style

Behavioural coding style describes the intended behavior of a circuit without specifying its implementation. In Verilog, this is achieved using procedural blocks, such as always blocks.

Example:

```
module combinational_logic(A, B, C);
  input [3:0] A, B;
  output [3:0] C;

  always @(*) begin
    C = A + B;
  end
endmodule
```

In the example above, the **combinational\_logic** module adds the inputs **A** and **B** and assigns the result to the output **C**. The **always @(\*)** block specifies that **C** is updated whenever **A** or **B** changes.

The behavioural coding style is useful for describing complex combinational logic and can be used for verification and simulation.

## 19. Synthesizable Coding Style

Synthesizable coding style describes the implementation of a circuit using hardware primitives that can be synthesized into physical gates. In Verilog, this is achieved using gate-level modelling or RTL modelling.

Example:

```
module combinational_logic(A, B, C);
  input [3:0] A, B;
  output [3:0] C;

  assign C = A + B;
endmodule
```

In the example above, the **combinational\_logic** module uses the **assign** statement to specify the implementation of the circuit. The **+** operator is used to add the inputs **A** and **B** and assign the result to the output **C**.

The synthesizable coding style is useful for implementing circuits on FPGAs or ASICs and can be used for synthesis and place-and-route. It is important to follow synthesizable coding guidelines and avoid constructs that cannot be synthesized into physical gates.

## 20. Behavioral Coding Style

Behavioral coding style describes the intended behavior of a circuit without specifying its implementation. In Verilog, this is achieved using procedural blocks, such as always blocks.

**Example:**

```
module sequential_logic(A, B, C, D);
    input A, B, C;
    output D;

    reg state;

    always @(posedge A) begin
        if (B && C) begin
            state <= 1'b1;
        end else begin
            state <= 1'b0;
        end
    end

    assign D = state;
endmodule
```

In the example above, the **sequential\_logic** module uses an always block triggered by the rising edge of input **A** to update the value of the **state** register. The **if** statement inside the always block specifies the behavior of the circuit. The output **D** is assigned the value of the **state** register.

The behavioral coding style is useful for describing complex sequential logic and can be used for verification and simulation.

## 21. Synthesizable Coding Style

Synthesizable coding style describes the implementation of a circuit using hardware primitives that can be synthesized into physical gates. In Verilog, this is achieved using state machines and flip-flops.

**Example:**

```

module sequential_logic(A, B, C, D);
    input A, B, C;
    output D;

    reg [1:0] state;
    always @(posedge A) begin
        case (state)
            2'b00: begin
                if (B && C) begin
                    state <= 2'b01;
                end else begin
                    state <= 2'b00;
                end
            end
            2'b01: begin
                state <= 2'b10;
            end
            2'b10: begin
                state <= 2'b00;
            end
        endcase
    end

    assign D = (state == 2'b01);
endmodule

```

In the example above, the **sequential\_logic** module uses a state machine to implement the circuit. The **state** register stores the current state of the machine, and the **case** statement inside the always block specifies the transitions between states. The output **D** is assigned the value of the **state** register.

The synthesizable coding style is useful for implementing circuits on FPGAs or ASICs and can be used for synthesis and place-and-route. It is important to follow synthesizable coding guidelines and avoid constructs that cannot be synthesized into physical gates.

## 22. Differences between behavioral and synthesizable coding styles for modelling combinational and sequential logic, along with examples

Coding Style	Combinational Logic Modeling	Sequential Logic Modeling	Example
Behavioral	Uses high-level constructs such as if-else statements and functions to describe the functionality of the circuit.	Uses flip-flops and other hardware-specific constructs to model the behavior of the circuit.	A behavioral model of a 4-bit adder would describe the functionality of the circuit in terms of the inputs and outputs, using if-else statements to describe the addition of two 4-bit numbers.
Synthesizable	Uses low-level constructs such as gates and flip-flops to describe the hardware implementation of the circuit.	Uses hardware-specific constructs such as registers and clock signals to model the behavior of the circuit.	A synthesizable model of a 4-bit adder would describe the circuit in terms of the gates and flip-flops used to implement the circuit, using registers to store the input and output values and a clock signal to synchronize the operation of the circuit.

Overall, the main difference between behavioral and synthesizable coding styles is that behavioral coding focuses on describing the functionality of the circuit using high-level constructs, while synthesizable coding focuses on describing the hardware implementation of the circuit using low-level constructs. Behavioral coding is typically easier to write and understand, but may not be suitable for synthesis into hardware. Synthesizable coding is more complex, but allows for a more accurate representation of the hardware implementation and is necessary for actual hardware implementation.

### **23. Behavioral and Synthesizable Coding Styles for Modelling Combinational Logic**

Introduction: In digital design, two coding styles are commonly used to model combinational logic: behavioral and synthesizable coding styles. Both coding styles can be used to describe the functionality of a digital circuit, but they differ in terms of their level of abstraction and the constructs used to describe the circuit.

Behavioral Coding Style: The behavioral coding style uses high-level constructs such as if-else statements and functions to describe the functionality of the circuit. It is more abstract and allows for a more concise and easy-to-understand description of the circuit. This style is used primarily for simulation and verification of the circuit, but not for actual hardware implementation.

Advantages:

- Easy to write and understand
- More concise and abstract
- Can be used for simulation and verification

Disadvantages:

- Not suitable for synthesis into hardware
- Not accurate in describing the hardware implementation

Examples:

- A behavioral model of a 4-bit adder would describe the functionality of the circuit in terms of the inputs and outputs, using if-else statements to describe the addition of two 4-bit numbers.

Synthesizable Coding Style: The synthesizable coding style uses low-level constructs such as gates and flip-flops to describe the hardware implementation of the circuit. It is more detailed and accurate in describing the circuit, and is necessary for actual hardware implementation.

Advantages:

- More accurate in describing the hardware implementation
- Suitable for synthesis into hardware
- Can be used for simulation and verification

Disadvantages:

- More complex and difficult to write and understand
- Less concise and more detailed

Examples:

- A synthesizable model of a 4-bit adder would describe the circuit in terms of the gates and flip-flops used to implement the circuit.

Conclusion: In conclusion, both behavioral and synthesizable coding styles can be used to model combinational logic, but they differ in terms of their level of abstraction and the constructs used to describe the circuit. Behavioral coding is more abstract and easier to understand, while synthesizable coding is more detailed and accurate in describing the hardware implementation. It is important to choose the appropriate coding style depending on the specific requirements of the design.

### **24. Behavioral and Synthesizable Coding Styles for Modelling Sequential Logic**

Introduction: In digital design, two coding styles are commonly used to model sequential logic: behavioral and synthesizable coding styles. Both coding styles can be used to describe the behavior of a digital circuit that depends on the history of its inputs and outputs, but they differ in terms of their level of abstraction and the constructs used to describe the circuit.

Behavioral Coding Style: The behavioral coding style uses high-level constructs such as if-else statements and functions to describe the behavior of the circuit. It is more abstract and allows for a more concise

and easy-to-understand description of the circuit. This style is used primarily for simulation and verification of the circuit, but not for actual hardware implementation.

Advantages:

- Easy to write and understand
- More concise and abstract
- Can be used for simulation and verification

Disadvantages:

- Not suitable for synthesis into hardware
- Not accurate in describing the hardware implementation

Examples:

- A behavioral model of a sequential circuit such as a counter would describe the behavior of the circuit in terms of the inputs and outputs, using if-else statements to describe the counting sequence and state transitions.

**Synthesizable Coding Style:** The synthesizable coding style uses low-level constructs such as registers, clock signals and flip-flops to describe the hardware implementation of the circuit. It is more detailed and accurate in describing the circuit, and is necessary for actual hardware implementation.

Advantages:

- More accurate in describing the hardware implementation
- Suitable for synthesis into hardware
- Can be used for simulation and verification

Disadvantages:

- More complex and difficult to write and understand
- Less concise and more detailed

Examples:

- A synthesizable model of a sequential circuit such as a counter would describe the circuit in terms of the registers, flip-flops and clock signals used to implement the circuit, and how they are interconnected to perform the counting and state transitions.

**Conclusion:** In conclusion, both behavioral and synthesizable coding styles can be used to model sequential logic, but they differ in terms of their level of abstraction and the constructs used to describe the circuit. Behavioral coding is more abstract and easier to understand, while synthesizable coding is more detailed and accurate in describing the hardware implementation. It is important to choose the appropriate coding style depending on the specific requirements of the design, taking into consideration the target platform and the level of detail required in the design.

**25. Table summarizing the differences between behavioural and synthesizable coding styles for modelling combinational and sequential logic, along with examples:**

	<b>Behavioral Coding Style</b>	<b>Synthesizable Coding Style</b>
<b>Modelling Combinational Logic</b>		
Focus	High-level behavior of the system	Low-level implementation of the system
Language	High-level languages, such as C, Python, and MATLAB	Hardware description languages, such as Verilog and VHDL
Constructs	High-level constructs, such as loops, conditionals, and arrays	Low-level constructs, such as logic gates and multiplexers
Example	<code>z = (a &amp; b)</code>	<code>(~a &amp; c);`</code>
Description	Describes the behavior of the combinational circuit	Describes the implementation of the circuit
Simulation Accuracy	More accurate in simulation, as it focuses on the high-level behavior of the system	Less accurate in simulation, as it focuses on the low-level implementation details
Implementation Effort	Lower implementation effort, as the code is closer to a human-readable specification of the system behavior	Higher implementation effort, as the code requires hardware-specific knowledge
<b>Modelling Sequential Logic</b>		
Focus	Behavior of the system over time	Low-level implementation of the system over time
Language	High-level languages, such as C, Python, and MATLAB	Hardware description languages, such as Verilog and VHDL
Constructs	High-level constructs, such as loops, conditionals, and functions	Low-level constructs, such as registers and flip-flops
Example	<b><code>always @(posedge clk) begin y &lt;= x; end</code></b>	<b><code>always @(posedge clk) begin y &lt;= x; end</code></b>
Description	Describes the behavior of the sequential circuit over time	Describes the implementation of the circuit over time
Simulation Accuracy	Accurate in simulation, as it describes the behavior of the sequential circuit over time	Accurate in simulation, as it describes the implementation of the circuit over time
Implementation Effort	Higher implementation effort, as the code requires an understanding of sequential circuits and their behavior over time	Lower implementation effort, as the code can be directly translated into hardware implementation