UNIT I

MOS TRANSISTOR THEORY

Basic MOSFET Structure

The cross-sectional and top/bottom view of MOSFET are as in figures 1 and 2 given below :

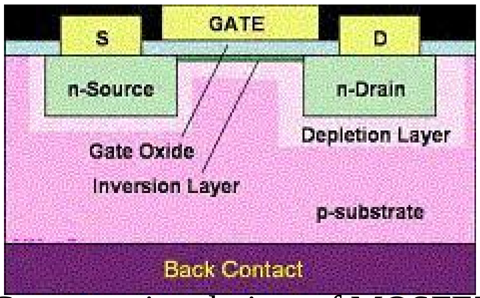


fig 1 Cross-sectional view of MOSFET

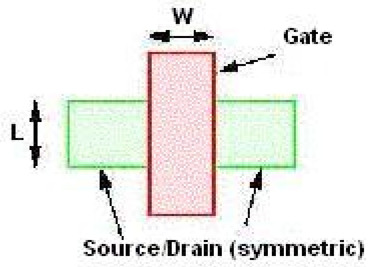
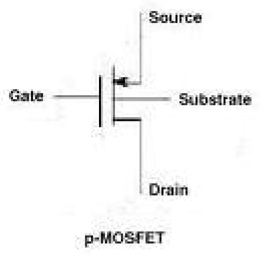


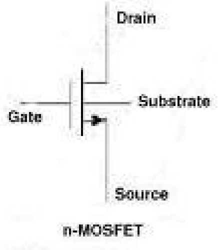
fig 2 Top/Bottom View of MOSFET An n-MOSFET

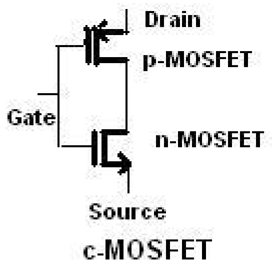
It consists of a **source** and a **drain**, two highly conducting n-type semiconductor regions which are separated from the p-type substrate by reverse-biased p-n diodes. A metal or poly crystalline gate covers the region between the source and drain, but is isolated from the semiconductor by the **gate oxide**.

Types of MOSFET

MOSFETs are divided into two types viz. **p-MOSFET** and **n-MOSFET** depending upon its type of source and drain.







p-MOSFET n-MOSFET c-MOSFET

The combination of a **n-MOSFET** and a **p-MOSFET** is called **cMOSFET** which is the mostly used as MOSFET transistor. We will look at it in more detail later.

MOSFET I-V Modelling

We are interested in finding the outputcharacteristics ( ) and the transfer charcteristics of the MOSFET. In other words, we can find out both if we can formulate a mathematical equation of the form:

$$I_{DS} = f(V_{DS}, V_{GS})$$

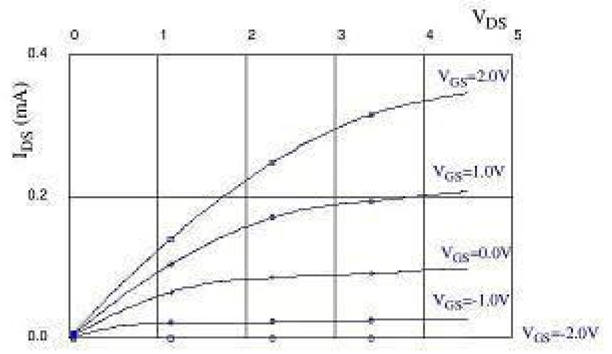
We can say that voltage level specifications and the material parameters cannot be altered by designers. So the only tools in the designer's hands with which he/she can improve the performance of the device are its dimensions, W and L In fact, the most important parameter in the device simulations is ratio of W and L.

The equations governing the **output** and **transfer** characteristics of an **n**-MOSFET and **p**-MOSFET are :

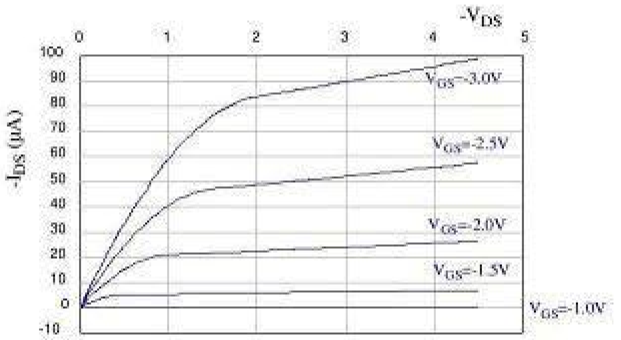
p-MOSFET:
$$I_{SD} = \begin{cases} 0.5\beta_p[2(V_{SG}-|V_{Tp}|)V_{SD} - V_{SD}^2] & \textbf{Linear} \\ 0.5\beta_p[V_{SG}-|V_{Tp}|]^2 & \textbf{Saturation} \end{cases}$$

n-MOSFET:
$$I_{DS} = \begin{cases} 0.5\beta_n[2(V_{GS}-|V_{Tn}|)V_{DS} - V_{DS}^2] & \textbf{Linear} \\ 0.5\beta_n[V_{GS}-|V_{Tn}|]^2 & \textbf{Saturation} \end{cases}$$

The **output** characteristics plotted for few fixed values of for **p**-MOSFET and **n**-MOSFET are shown next :

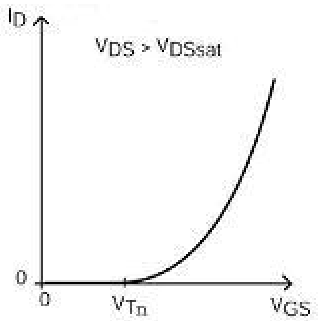


p-MOSFET

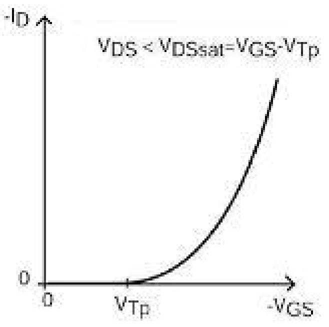


n-MOSFET

The **transfer** characteristics of both **p**-MOSFET and **n**-MOSFET are plotted for a fixed value of as shown next :



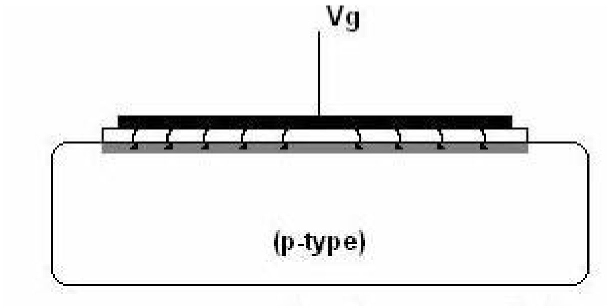
p-MOSFET



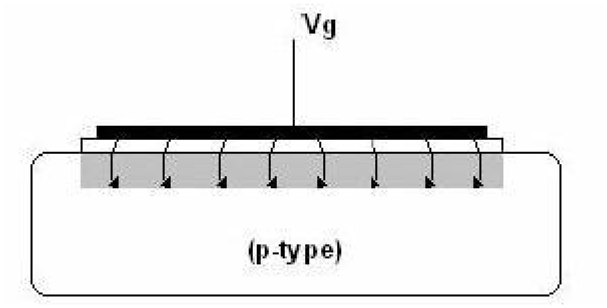
n-MOSFET

Modes of operation

Depending upon the value of gate voltage applied, the MOS capacitor works in three modes :



Accumulation mode (grey layer - strong hole concentration)



Depletion Mode (light grey layer – depletion region)

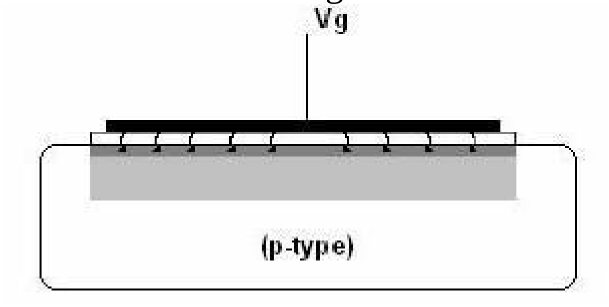
1. **Accumulation:** In this mode, there is accumulation of holes (assuming n-MOSFET) at the Si-SiO2 interface. All the field lines emanating from the gate terminate on this layer giving an effective dielectric thickness as the oxide thickness. In this mode, Vg <0

2. **Depletion:** As we move from negative to positive gate voltages the

holes at the interface are repelled and pushed back into the bulk leaving a depleted layer. This layer counters the positive charge on the gate and keeps increasing till the gate voltage is below threshold voltage. we see a larger effective dielectric length and hence a lower capacitance.

$\frac{kT}{} ln(\frac{N_A}{n_i})$

3. **Strong Inversion:**When Vg crosses threshold voltage, the increase in depletion region width stops and charge on layer is countered by mobile electrons at Si-SiO2 interface. This is called inversion because the mobile charges are opposite to the type of charges found in substrate. In this case the inversion layer is formed by the electrons. Field lines hence terminate on this layer thereby reducing the effective dielectric thickness



Strong Inversion mode
(grey layer - strongelectron concentration, light grey - depletion region)

Threshold voltage

It is that gate voltage at which the surface band bending is twice φ_F, Where $= q \varphi_F$

We know that the depth of depletion region for φ_s is between **0** and $2\varphi_F$ and is given by,

$$x_d = \sqrt{\frac{2 \in_s \varphi_s}{qN_a}}$$

Charge in depletion region at $\varphi_s = 2\varphi_F$ is given by $Q_B = -qN_a x_{dmax}$ where

$$x_{dmax} = \sqrt{\frac{2 \in_s (2\varphi_F)}{qN_a}}$$

Beyond threshold, the total charge **Q**D in the seminconductor has to balance the charge on gate electrode, **Qs** i.e. $Q_s = -(Q_I + Q_D)$ where we define the charge in the inversion layer as a quantity which needs to be determined.

This leads to following expression for gate voltage-

$$V_{GS} \quad V_{FB} + \varphi_s - \frac{Q_S}{C_{ox}} = V_{FB} + \varphi_s - \frac{(Q_i + Q_D)}{C_{ox}}$$

In case of depletion, there in no inversion layer charge, so **Qi** =**0**, i.e. gate voltage becomes

$$V_{GS} \quad V_{FB} + \varphi_s - \frac{Q_D}{C_{ox}} = V_{FB} + \varphi_s + \frac{2\sqrt{qN_a \in_s \varphi_F}}{C_{ox}} \text{ for } 0 \leq \varphi_S \leq 2\varphi_F$$

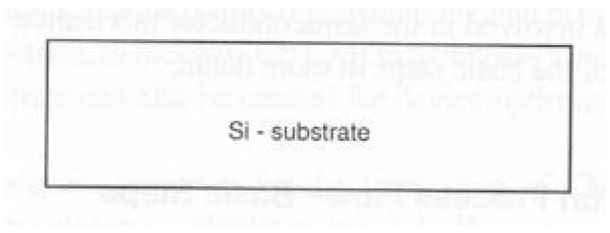
but in case of inversion, the gate voltage will be given by :

The second term in second equality of last expression states our basic assumption, namely that any change in gate voltage beyond the threshold requires a change in inversion layer charge. Also from the same expression, we obtain threshold voltage as :

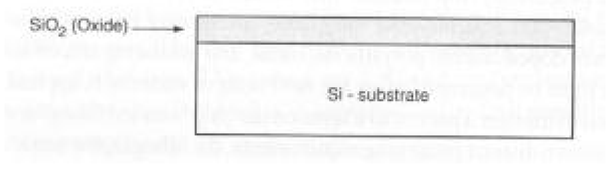
$$= V_{FB} + 2V_T + \frac{2\sqrt{qN_a \in_s \varphi_F}}{C_{ox}}$$

MOS Fabrication:

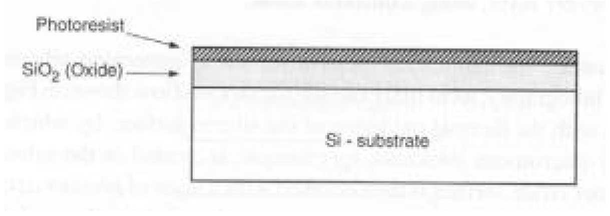
Step1:
Processing is carried on single crystal silicon of high purity on which required P impurities are
introduced as crystal is grown. Such wafers are about 75 to 150 mm in diameter and 0.4 mm thick and they are doped with say boron to impurity concentration of 10 to power 15/cm3 to 10 to the power 16 /cm3.



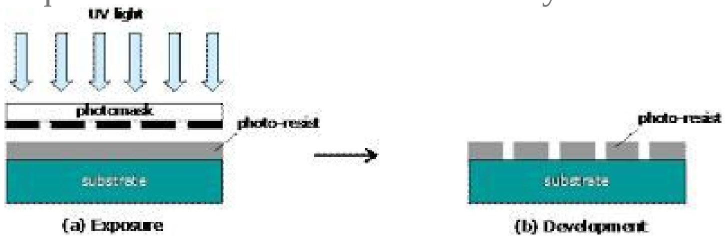
Step 2 :
A layer of silicon di oxide (SiO2) typically 1 micrometer thick is grown all over the surface of
the wafer to protect the surface, acts as a barrier to the dopant during processing, and provide a generally insulating substrate on to which other layers may be deposited and patterned.



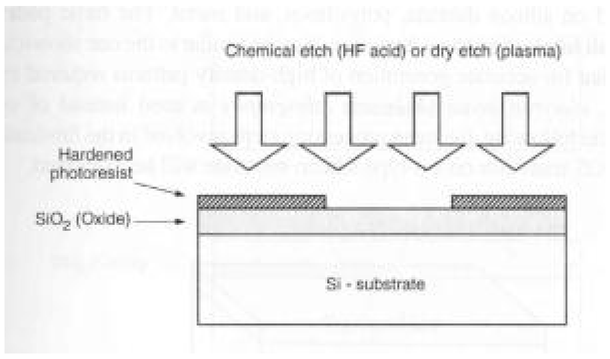
Step 3:
The surface is now covered with the photo resist which is deposited onto the wafer and spun to an
even distribution of the required thickness.



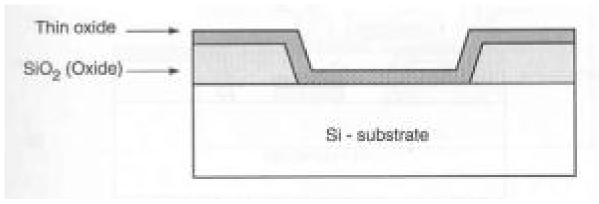
Step 4:
The photo resist layer is then exposed to ultraviolet light through masking which defines those
regions into which diffusion is to take place together with transistor channels. Assume, for example , that those areas exposed to uv radiations are polymerized (hardened), but that the areas
required for diffusion are shielded by the mask and remain unaffected.



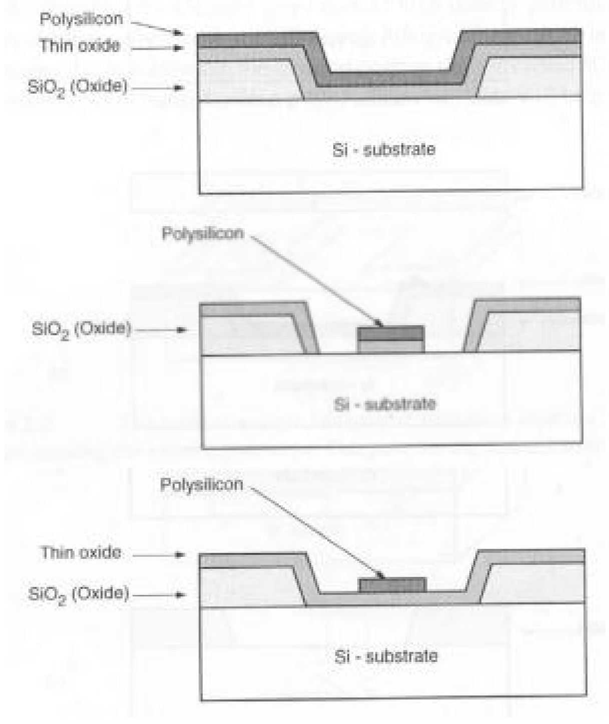
Step 5:
These areas are subsequently readily etched away together with the underlying silicon di oxide so
that the wafer surface is exposed in the window defined by the mask.



Step 6:
The remaining photo resist is removed and a thin layer of SiO2 (0.1 micro m typical) is grown
over the entire chip surface and then poly silicon is deposited on the top of this to form the gate structure. The polysilicon layer consists of heavily doped polysilicon deposited by chemical
vapour deposition (CVD). In the fabrication of fine pattern devices, precise control of thickness, impurity concentration, and resistivity is necessary

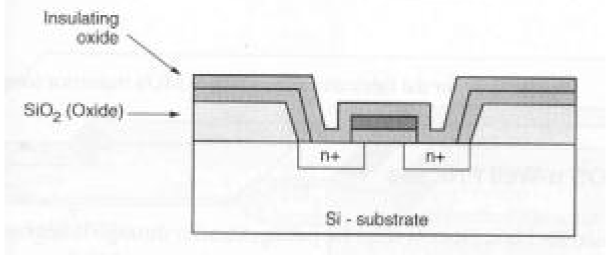


Step 7:
Further photo resist coating and masking allows the poly silicon to be patterned and then the thin
oxide is removed to expose areas into which n-type impurities are to be diffused to form the source and drain. Diffusion is achieved by heating the wafer to a high temperature and passing a gas containing the desired n-type impurity.
Note: The poly silicon with underlying thin oxide and the thick oxide acts as mask during
diffusion the process is self aligning.



Step 8:

Thick oxide (SiO2) is grown over all again and is then masked with photo resist and etched to expose selected areas of the poly silicon gate and the drain and source areas where connections are to be made. (contacts cut)



Step 9:
The whole chip then has metal (aluminium) deposited over its surface to a thickness typically of 1
micro m. This metal layer is then masked and etched to form the required interconnection pattern.

CMOS Fabrication:

CMOS fabrication can be accomplished using either of
the three technologies:

- N-well/P-well technologies
- Twin well technology
- Silicon On Insulator (SOI)

Twin Well Technology

Using twin well technology, we can optimise NMOS and PMOS transistors separately. This means that transistor parameters such as threshold voltage, body effect and the channel transconductance of both types of transistors can be tuned independenly.

n+ or p+ substrate, with a lightly doped epitaxial layer on top, forms the starting material for this technology. The n-well and pwell are formed on this epitaxial layer which forms the actual substrate. The dopant concentrations can be carefully optimized to produce the desired device characterisitcs because two independent doping steps are performed to create the well regions.

The conventional n-well CMOS process suffers from, among other effects, the problem of unbalanced drain parasitics since the doping density of the well region typically being about one order of magnitude higher than the substrate. This problem is absent in the twin-tub process.

Silicon on Insulator (SOI)

To improve process characteristics such as speed and latch-up susceptibility, technologists have sought to use an insulating substrate instead of silicon as the substrate material.

Completely isolated NMOS and PMOS transistors can be created virtually side by side on an insulating substrate (eg. sapphire) by using the SOI CMOS technology.

This technology offers advantages in the form of higher integration density (because of the absence of well regions), complete avoidance of the latch-up problem, and lower parasitic capacitances compared to the conventional n-well or twin-tub CMOS processes.

But this technology comes with the disadvantage of higher cost than the standard n-well CMOS process. Yet the improvements of device performance and the absence of latch- up problems can justify its use, especially in deep submicron devices.

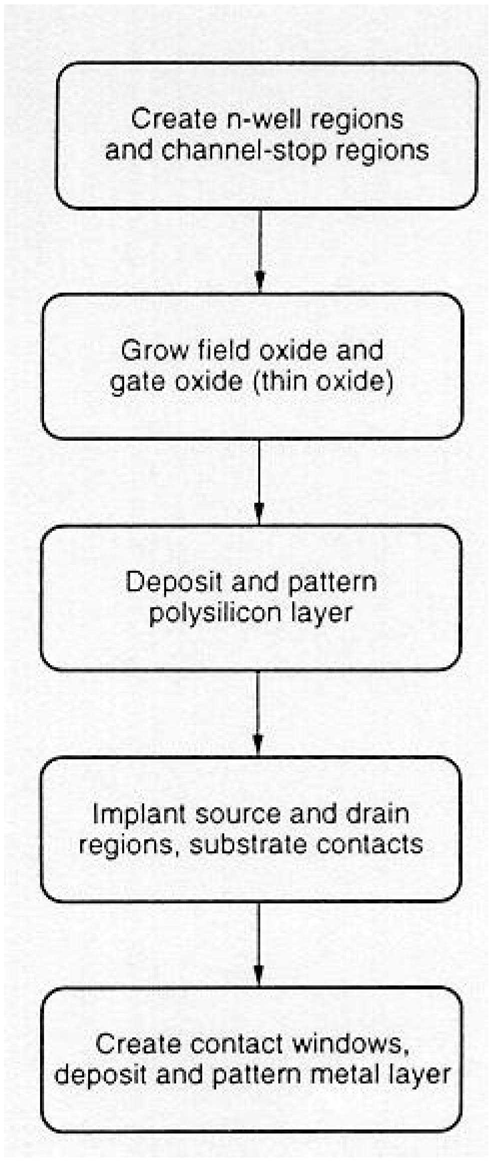
In this discussion we will concentrate on the well established n-well CMOS fabrication technology, which requires that both nchannel and p-channel transistors be built on the same chip substrate. To accomodate this, special regions are created with a semiconductor type opposite to the substrate type. The regions thus formed are called wells or tubs. In an n-type substrate, we can create a p-well or alternatively, an n-well is created in a p-type substrate. We present here a simple n-well CMOS fabrication technology, in which the NMOS transistor is created in the p-type substrate, and the PMOS in the n-well, which is built-in into the p-type substrate.

Historically, fabrication started with p-well technology but now it has been completely shifted to n-well technology. The main reason for this is that, "n-well sheet resistance can be made lower than p-well sheet resistance" (electrons are more mobile than holes).

The simplified process sequence (shown in Figure 12.41) for the fabrication of CMOS
integrated circuits on a p-type silicon
substrate is as follows:

- N-well regions are created for PMOS transistors, by impurity implantation into the substrate.
- This is followed by the growth of a thick oxide in the regions surround the NMOS and PMOS active regions.
- The thin gate oxide is subsequently grown on the surface through thermal oxidation.
- After this n+ and p+ regions (source, drain and channel-stop implants) are created.

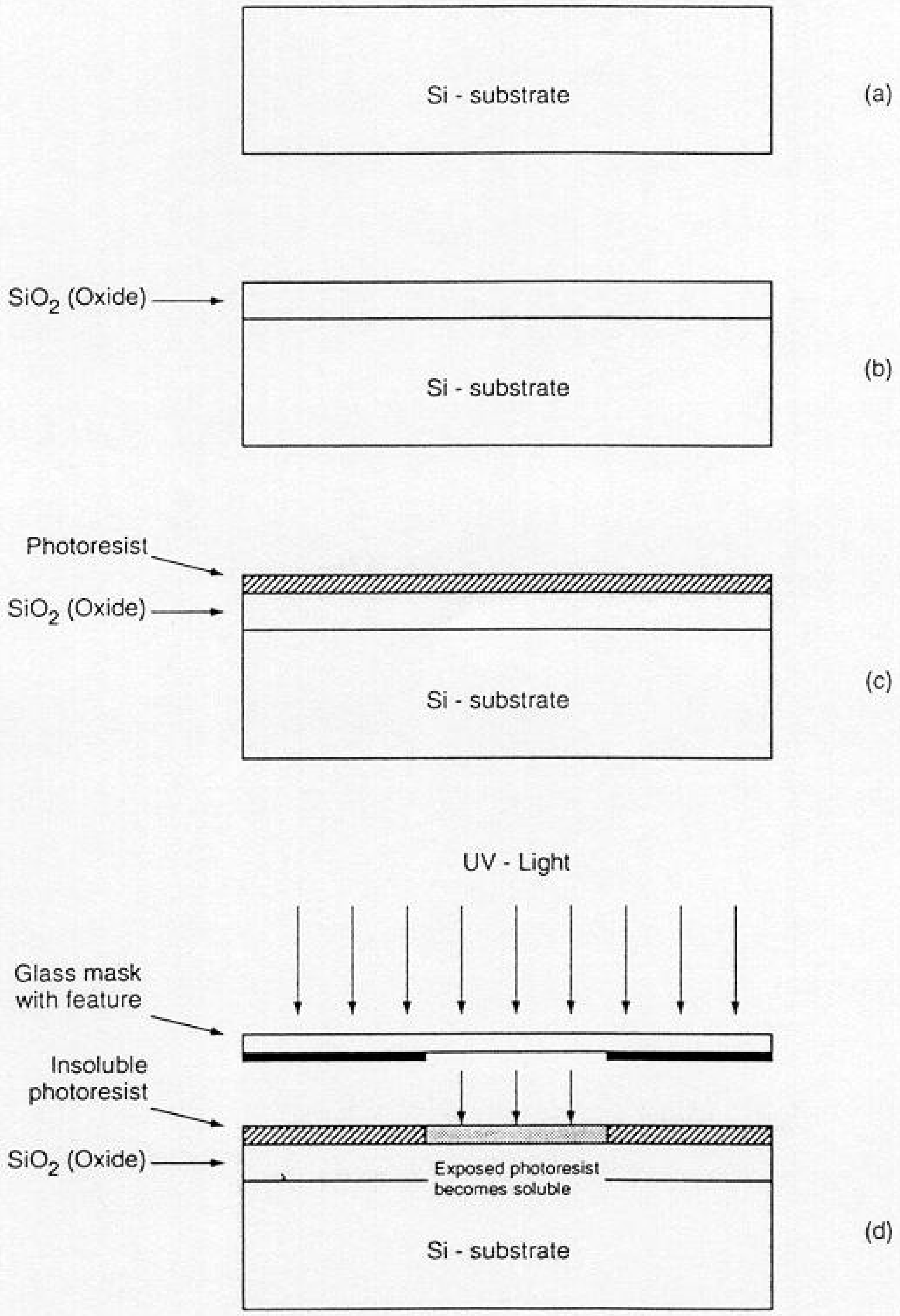
- The metallization step (creation of metal interconnects) forms the final step in this
  process.

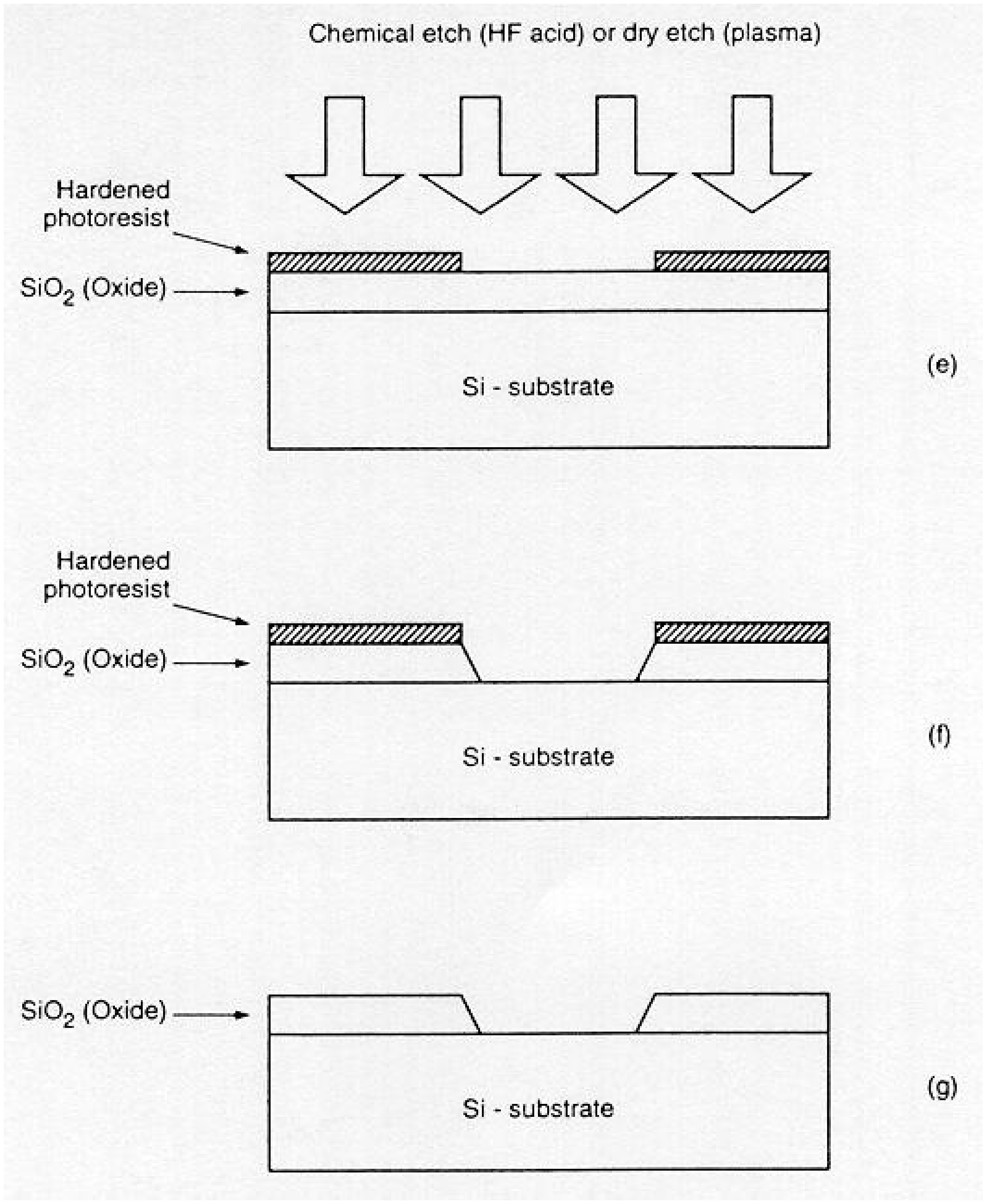

Create n-well regions and channel-stop regions
Grow field oxide and gate oxide (thin oxide)
Deposit and pattern polysilicon layer
Implant source and drain regions, substrate contacts
Create contact windows, deposit and pattern metal layer

Simplified Process Sequence For Fabrication Of CMOS ICs

The integrated circuit may be viewed as a set of patterned layers of doped silicon, polysilicon, metal and insulating silicon dioxide, since each processing step requires that certain areas are defined on chip by appropriate masks. A layer is patterned before the next layer of material is applied on the chip. A process, called lithography, is used to transfer a pattern to a layer. This must be repeated for every layer, using a different mask, since each layer has its own distinct requirements.

We illustrate the fabrication steps involved in patterning silicon dioxide through optical lithography, which shows the lithographic sequences.


Si - substrate
(a)
SiO$_2$ (Oxide)
Si - substrate
(b)
Photoresist
SiO$_2$ (Oxide)
Si - substrate
(c)
UV - Light
Glass mask
with feature
Insoluble
photoresist
SiO$_2$ (Oxide)
Exposed photoresist
becomes soluble
Si - substrate
(d)



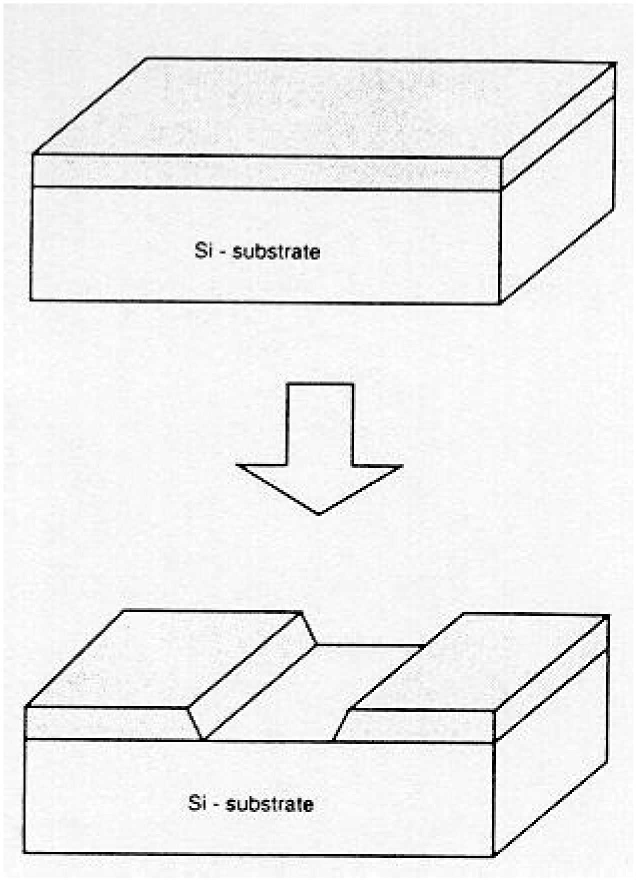
Process steps required for patterning of silicon dioxide

First an oxide layer is created on the substrate with thermal oxidation of the silicon surface. This oxide surface is then covered with a layer of photoresist. Photoresist is a light-sensitive, acid-resistant organic polymer which is initially insoluble in the developing solution. On exposure to ultraviolet (UV) light, the exposed areas become soluble which can be etched away by etching solvents. Some areas on the surface are covered with a mask during exposure to selectively expose the photoresist. On exposure to UV light, the masked areas are shielded whereas those areas which are not shielded become soluble.

There are two types of photoresists, positive and negative photoresist. Positive photoresist is initially insoluble, but becomes soluble after exposure to UV light, where as negative photoresist is initially soluble but becomes insoluble (hardened) after exposure to UV light. The process sequence described uses positive photoresist.

Negative photoresists are more sensitive to light, but their photolithographic resolution is not as high as that of the positive photoresists. Hence, the use of negative photoresists is less common in manufacturing high-density integrated circuits.

The unexposed portions of the photoresist can be removed by a solvent after the UV exposure step. The silicon dioxide regions not covered by the hardened photoresist is etched away by using a chemical solvent (HF acid) or dry etch (plasma etch) process. On completion of this step, we are left with an oxide window which reaches down to the silicon surface. Another solvent is used to strip away the remaining photoresist from the silicon dioxide surface. The patterned silicon dioxide feature is shown in Figure 12.43



The result of single photolithographic patterning sequence on silicon dioxide

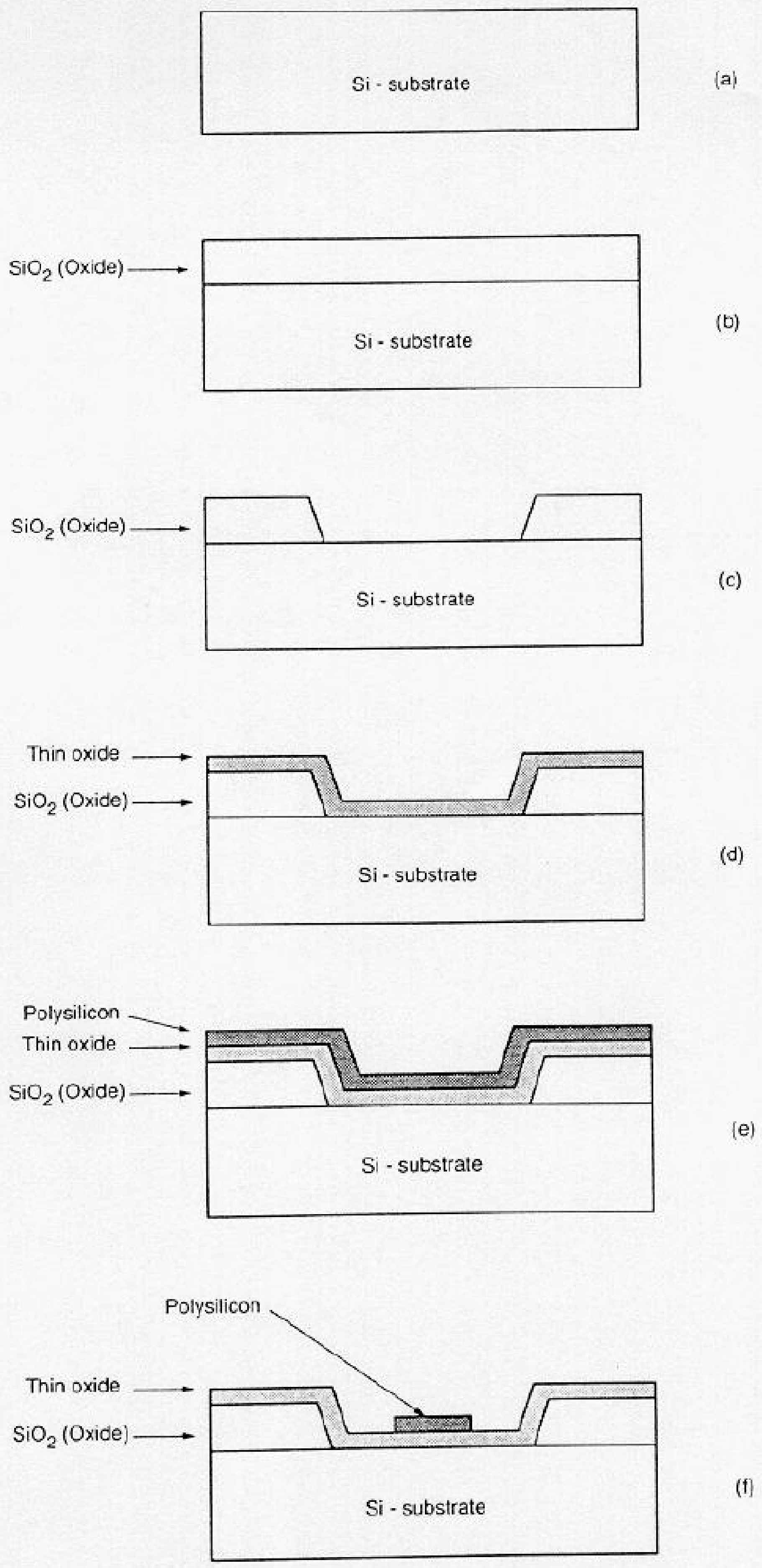
The sequence of process steps illustrated in detail actually accomplishes a single pattern transfer onto the silicon dioxide surface. The fabrication of semiconductor devices requires several such pattern transfers to be performed on silicon dioxide, polysilicon, and metal. The basic patterning process used in all fabrication steps, however, is quite similar to the one described earlier. Also note that for accurate generation of high- density patterns required in submicron devices, electron beam (E-beam) lithography is used instead of optical lithography.

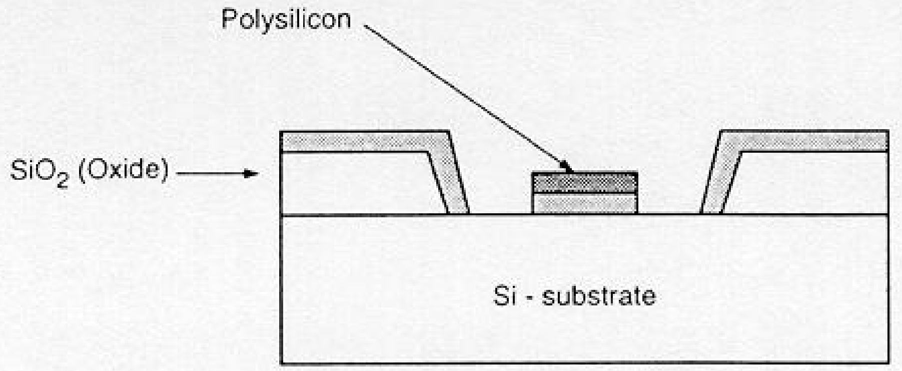
In this section, we will examine the main processing steps involved in fabrication of an n-channel MOS transistor on a p-type silicon substrate.

The first step of the process is the oxidation of the silicon substrate which creates a relatively thick silicon dioxide layer on the surface. This oxide layer is called field oxide The field oxide is then selectively etched to expose the silicon surface on which the transistor will be created. After this the surface is covered with a thin, high-quality oxide layer. This oxide layer will form the gate oxide of the MOS transistor Then a polysilicon layer is deposited on the thin oxide Polysilicon is used as both a gate electrode material for MOS transistors as well as an interconnect medium in silicon integrated circuits. The resistivity of polysilicon, which is usually high, is reduced by doping it with impurity atoms.

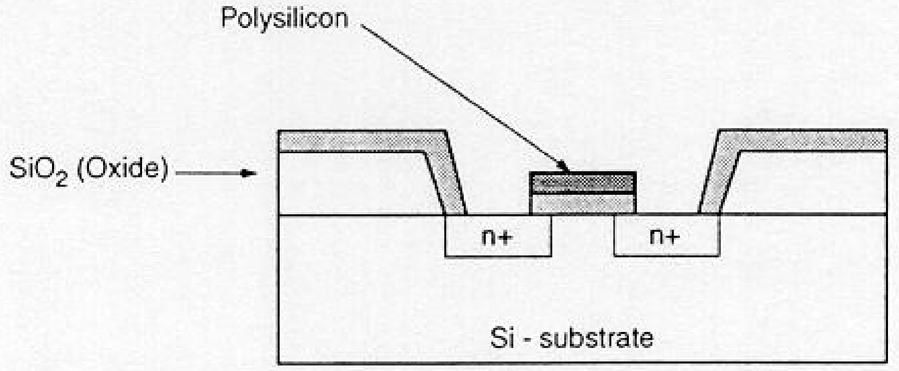
Deposition is followed by patterning and etching of polysilicon layer to form the interconnects and the MOS transistor gates The thin gate oxide not masked by polysilicon is also etched away exposing the bare silicon surface. The drain and source junctions are to be formed Diffusion or ion implantation is used to dope the entire silicon surface with a high concentration of impurities (in this case donor atoms to produce n-type doping). Two n-type regions (source and drain junctions) in the p-type substrate as doping penetrates the exposed areas of the silicon surface. The penetration of impurity doping into the polysilicon reduces its resistivity. The polysilicon gate is patterned before the doping and it precisely defines the location of the channel region and hence, the location of the source and drain regions. Hence this process is called a self-aligning process.

The entire surface is again covered with an insulating layer of silicon dioxide after the source and drain regions are completed Next contact windows for the source and drain are patterned into the oxide layer . Interconnects are formed by evaporating aluminium on the surface which is followed by patterning and etching of the metal layer A second or third layer of metallic interconnect can also be added after adding another oxide layer, cutting (via) holes, depositing and patterning the metal.

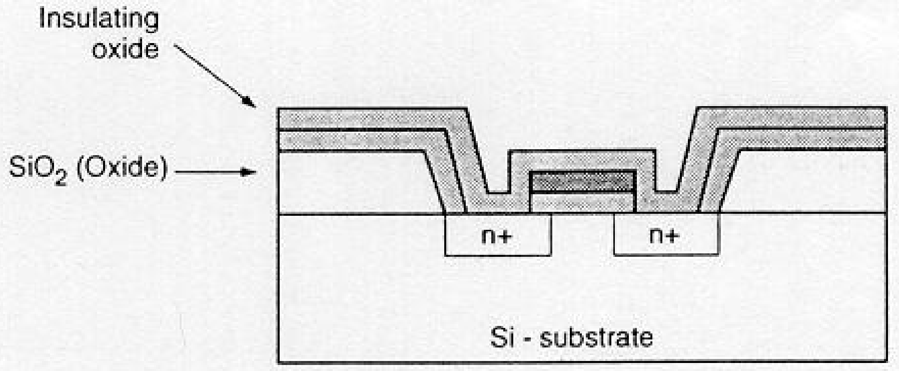

Si - substrate
(a)
SiO$_2$ (Oxide)
Si - substrate
(b)
SiO$_2$ (Oxide)
Si - substrate
(c)
Thin oxide
SiO$_2$ (Oxide)
Si - substrate
(d)
Polysilicon
Thin oxide
SiO$_2$ (Oxide)
Si - substrate
(e)
Polysilicon
Thin oxide
SiO$_2$ (Oxide)
Si - substrate
(f)


Polysilicon
SiO$_2$ (Oxide)
Si - substrate

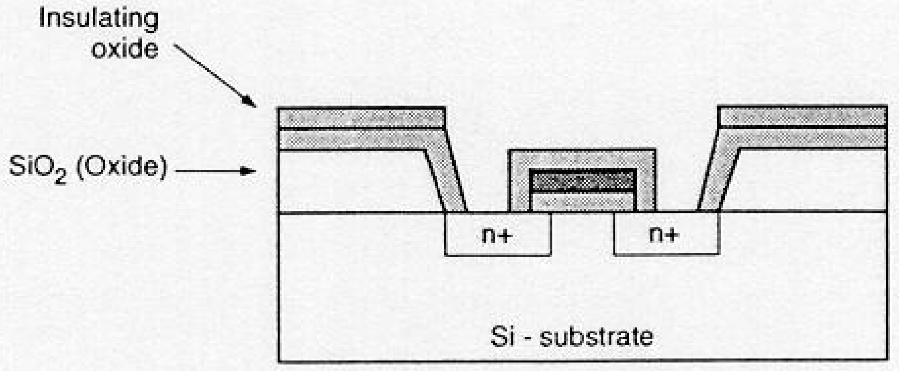
(g)


Polysilicon
SiO$_2$ (Oxide)
n+
n+
Si - substrate

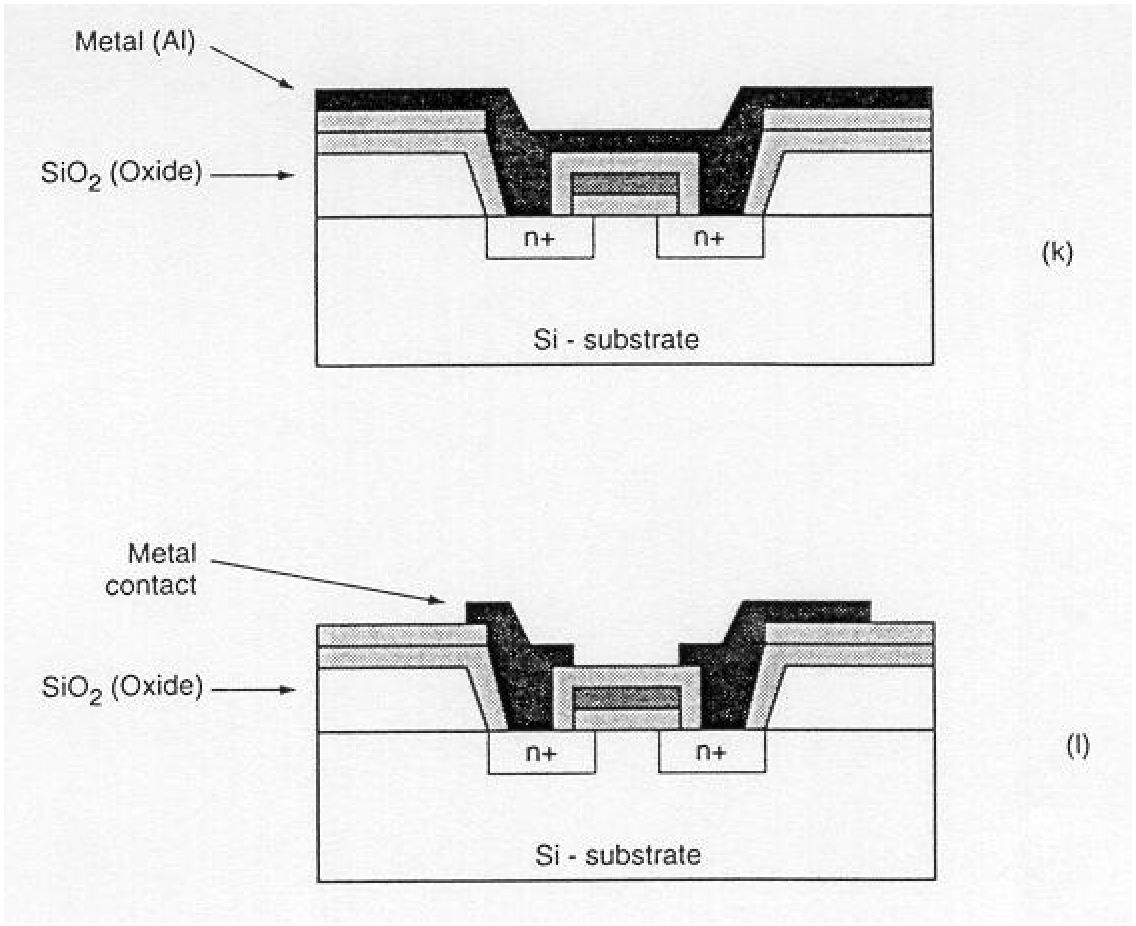
(h)


Insulating oxide
SiO$_2$ (Oxide)
n+
n+
Si - substrate

(i)


Insulating oxide
SiO$_2$ (Oxide)
n+
n+
Si - substrate

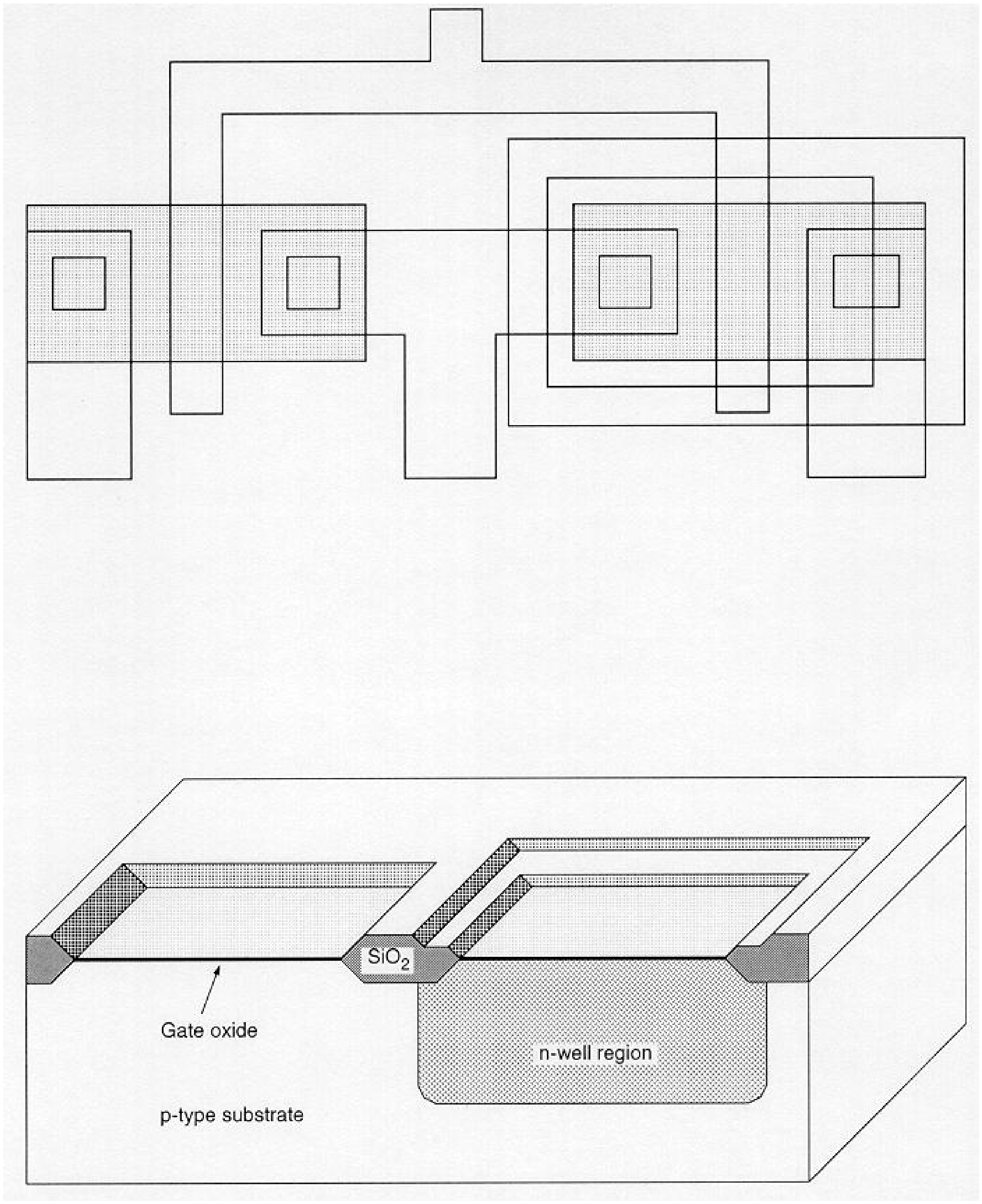
(j)



Process flow for the fabrication of an n-type MOSFET on p-type silicon

We now return to the generalized fabrication sequence of n-well CMOS integrated circuits. The following figures illustrate some of the important process steps of the fabrication of a CMOS inverter by a top view of the lithographic masks and a cross- sectional view of the relevant areas.

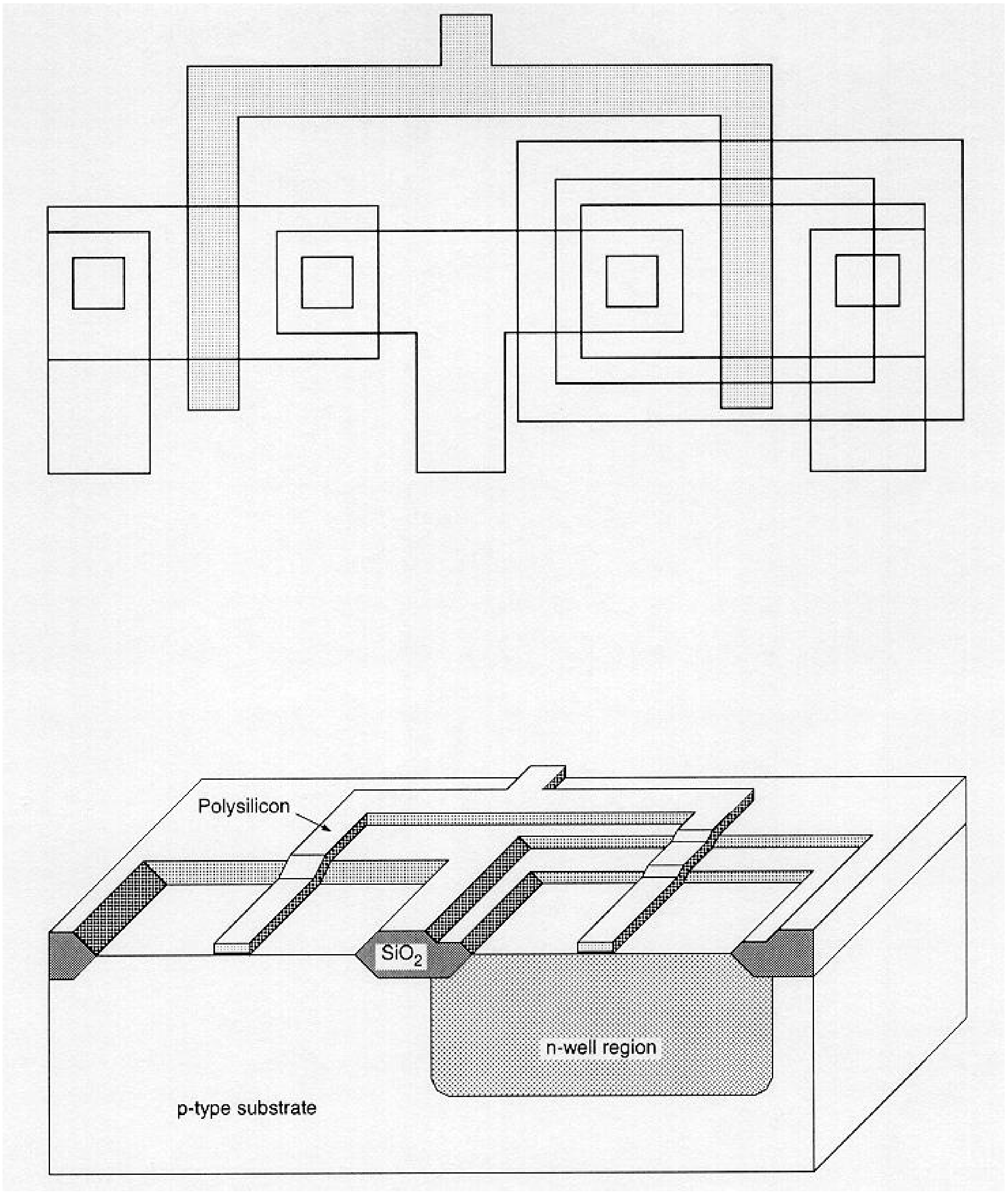
The n-well CMOS process starts with a moderately doped (with impurity concentration typically less than 1015 cm-3) p-type silicon substrate. Then, an initial oxide layer is grown on the entire surface. The first lithographic mask defines the n-well region. Donor atoms, usually phosphorus, are implanted through this window in the oxide. Once the n- well is created, the active areas of the nMOS and pMOS transistors can be defined



The creation of the n-well region is followed by the growth of a thick field oxide in the areas surrounding the transistor active regions, and a thin gate oxide on top of the active regions. The two most important critical fabrication parameters are the thickness and quality of the gate oxide. These strongly affect the operational characteristics of the MOS transistor, as well as its long-term stability.

Chemical vapor deposition (CVD) is used for deposition of polysilicon layer and patterned by dry (plasma) etching. The resulting polysilicon lines function as the gate electrodes of the nMOS and the pMOS transistors and their interconnects. The polysilicon gates also act as self-aligned masks for source and drain implantations.

The n+ and p+ regions are implanted into the substrate and into the n-well using a set of two masks. Ohmic contacts to the substrate and to the n-well are also implanted in this process step.

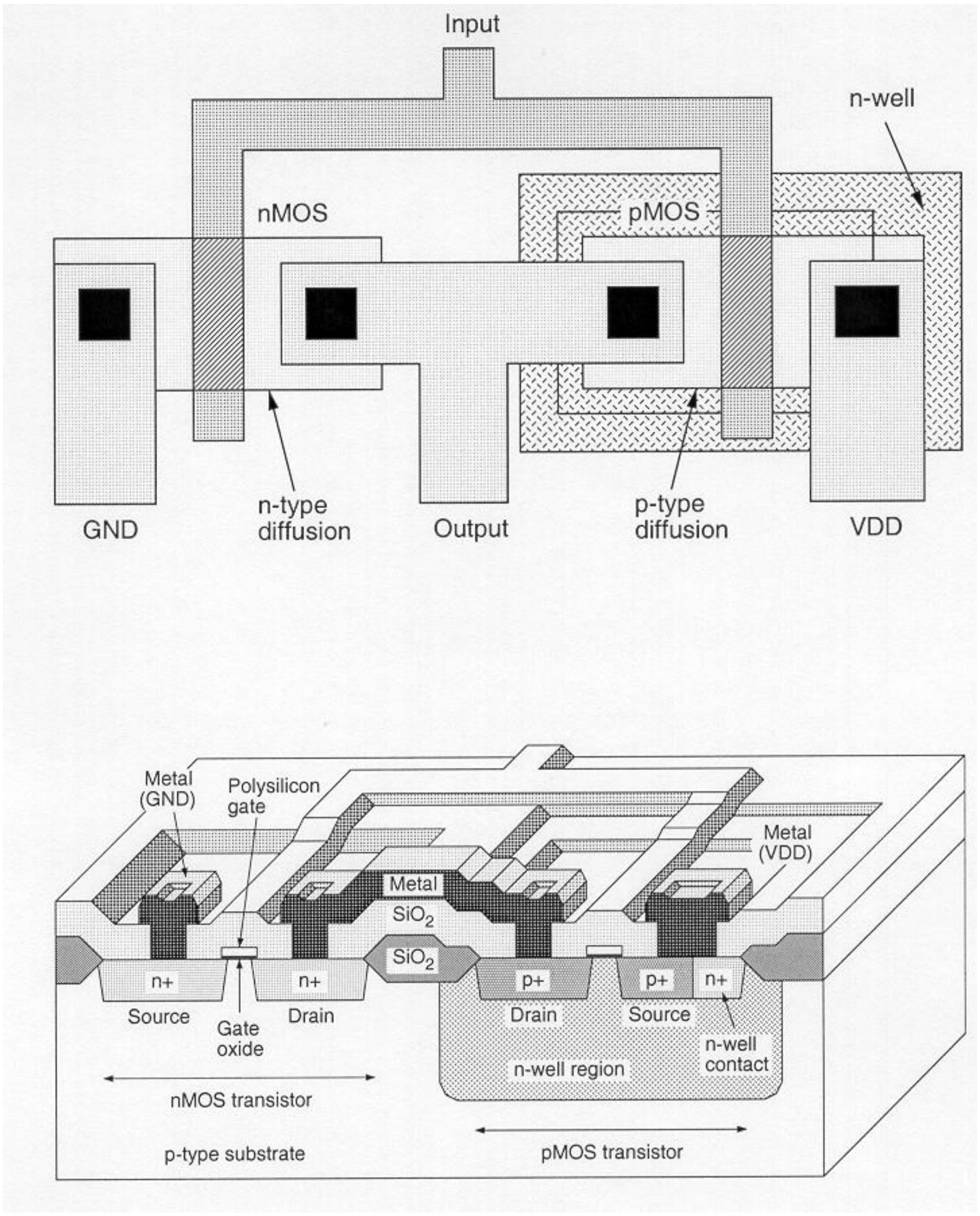


CVD is again used to deposit and insulating silicon dioxide layer over the entire wafer. After this the contacts are defined and etched away exposing the silicon or polysilicon contact windows. These contact windows are essential to complete the circuit interconnections using the metal layer, which is patterned in the next step.

Metal (aluminum) is deposited over the entire chip surface using metal evaporation, and the metal lines are patterned through etching. Since the wafer surface is non-planar, the quality and the integrity of the metal lines created in this step are very critical and are ultimately essential for circuit reliability.

The composite layout and the resulting cross-sectional view of the chip, showing one nMOS and one pMOS transistor (built-in nwell), the polysilicon and metal interconnections. The final step is to deposit the passivation layer (for protection) over the chip, except for wire-bonding pad areas.

This completes the fabrication of the CMOS inverter using n-well technology.



UNIT II

MOS CIRCUITS AND DESIGN

MOSFET I-V Modelling

We are interested in finding the outputcharacteristics ( ) and the transfer charcteristics of the MOSFET. In other words, we can find out both if we can formulate a mathematical equation of the form:

$$I_{DS} = f(V_{DS}, V_{GS})$$

We can say that voltage level specifications and the material parameters cannot be altered by designers. So the only tools in the designer's hands with which he/she can improve the performance of the device are its dimensions, W and L In fact, the most important parameter in the device simulations is ratio of W and L.

The equations governing the **output** and **transfer** characteristics of an **n**-MOSFET and **p**-MOSFET are :

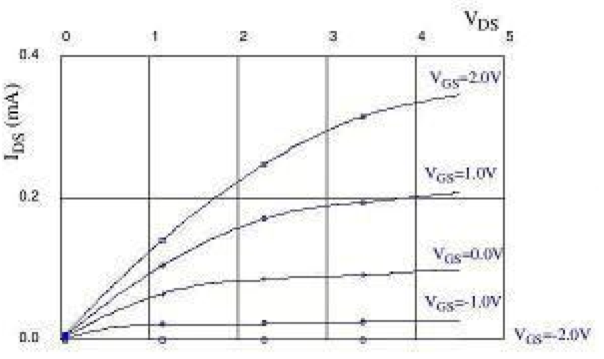
p-MOSFET:

$$I_{SD} = \begin{cases} 0.5\beta_p[2(V_{SG} - |V_{Tp}|)V_{SD} - V_{SD}^2] & \textbf{Linear} \\ 0.5\beta_p[V_{SG} - |V_{Tp}|]^2 & \textbf{Saturation} \end{cases}$$

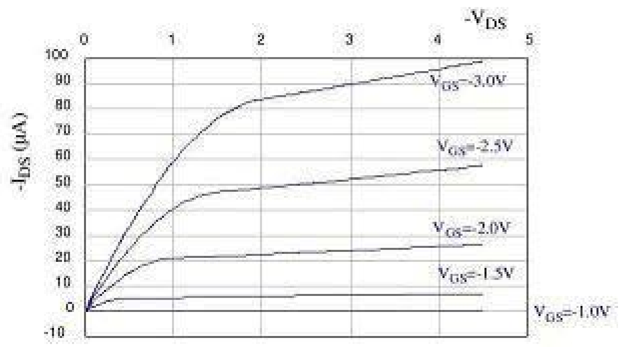
n-MOSFET:

$$I_{DS} = \begin{cases} 0.5\beta_n[2(V_{GS} - |V_{Tn}|)V_{DS} - V_{DS}^2] & \textbf{Linear} \\ 0.5\beta_n[V_{GS} - |V_{Tn}|]^2 & \textbf{Saturation} \end{cases}$$

The **output** characteristics plotted for few fixed values of for **p**-MOSFET and **n**-MOSFET are shown next :

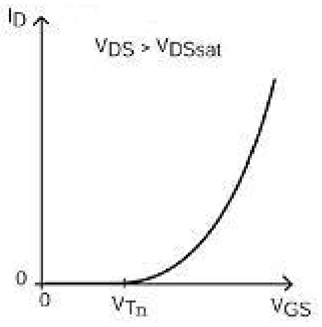


p-MOSFET

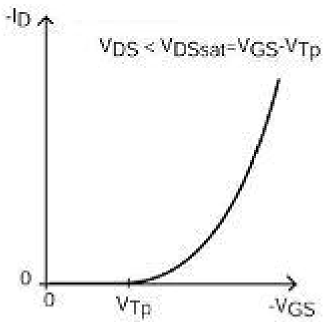


n-MOSFET

The **transfer** characteristics of both **p**-MOSFET and **n**-MOSFET are plotted for a fixed value of as shown next :



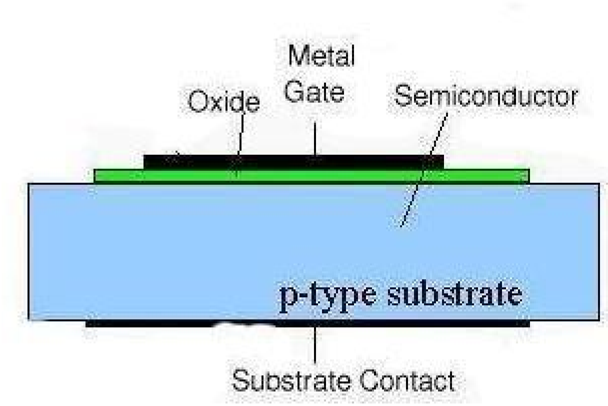
p-MOSFET



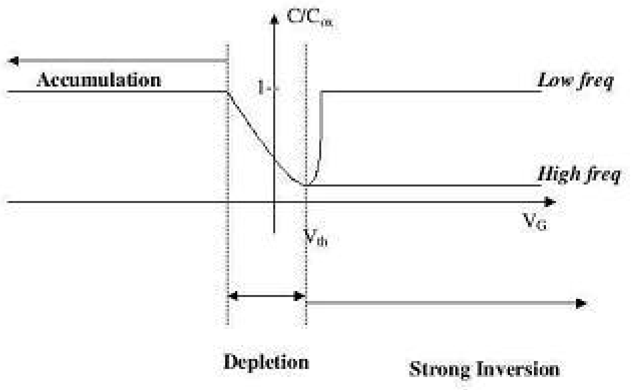
n-MOSFET

C-V Characteristics of a MOS Capacitor

As we have seen earlier, there is an oxide layer below Gate terminal. Since oxide is a very good insulator, it contributes to an oxide capacitance in the circuit. Normally, the capacitance value of a capacitor doesn't change with values of voltage applied across its terminals. However, this is not the case with MOS capacitor. We find that the capacitance of MOS capacitor changes its value with the variation in Gate voltage. This is because application of gate voltage results in the band bending of silicon substrate and hence variation in charge concentration at **Si-SiO2** interface. Also we can see that the curve splits into two (reason will be explained later), after a certain voltage, depending upon the frequency (high or low) of AC voltage applied at the gate. This voltage is called the threshold voltage(**Vth**) of MOS capacitor.



Cross section view of MOS Capacitor



plot of MOS Capacitor

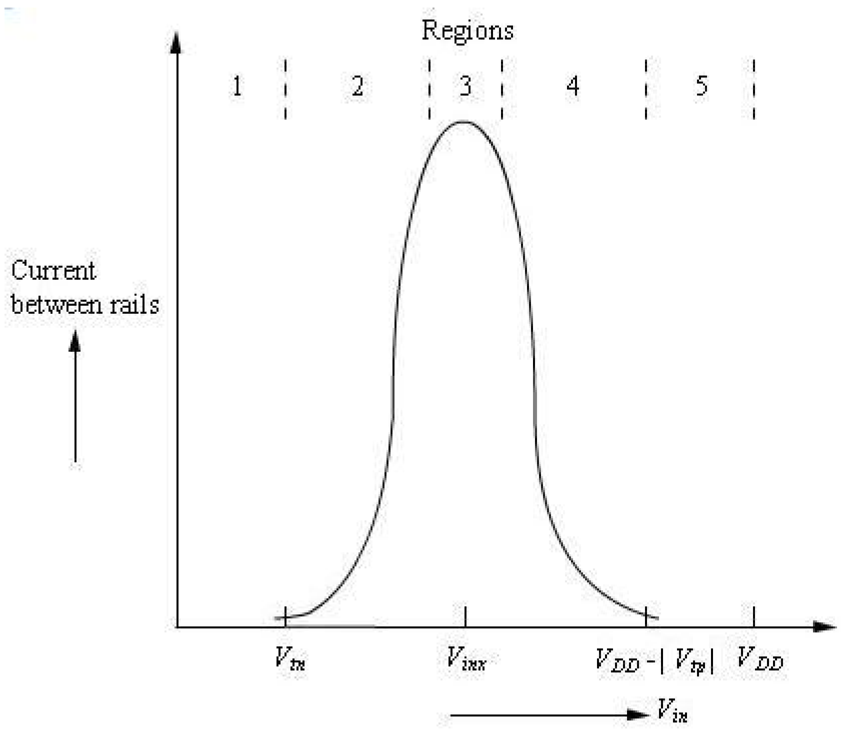
DC Characteristics of CMOS:

Let V_{tn} and V_{tp} denote the threshold voltages of the *n* and *p*-devices respectively. The following voltages at the gate and the drain of the two devices (relative to their respective sources) are all referred with respect to the ground (or V_{SS}), which is the substrate voltage of the *n* -device, namely

$V_{gsn} = V_{in}$, $V_{dsn} = V_{out}$, $V_{gsp} = V_{in} - V_{DD}$, and $V_{dsp} = V_{out} - V_{DD}$.

The voltage transfer characteristic of the CMOS inverter is now derived with reference to the following five regions of operation :

Region 1 : the input voltage is in the range $0 \le V_{in} < V_{tn}$. In this condition, the *n* -transistor is off, while the *p* -transistor is in linear region (as $-V_{DD} < V_{gsp} < -V_{DD} + V_{tn}$).



No actual current flows until V_{in} crosses V_{tn} , as may be seen from Figure 2.11. The operating point of the *p* -transistor moves from higher to lower values of currents in linear zone.

Region 2 : the input voltage is in the range $V_{tn} \le V_{in} < V_{inv}$. The upper limit of V_{in} is V_{inv} , the *logic threshold voltage* of the inverter. The logic threshold voltage or the *switching point voltage* of an inverter denotes the boundary of "logic 1" and "logic 0". It is the output voltage at which $V_{in} = V_{out}$. In this region, the *n*-transistor moves into saturation, while the *p*-transistor remains in linear region. The total current through the inverter increases, and the output voltage tends to drop fast.

Region 3 : In this region, $V_{in} \approx V_{inv}$. Both the transistors are in saturation, the drain current attains a maximum value, and the output voltage falls rapidly. The inverter exhibits gain. But this region is inherently unstable. As both the transistors are in saturation, equating their currents, one gets

(as $V_{gsn} = V_{inv}$, $V_{gsp} = V_{inv} - V_{DD}$).

$$\frac{1}{2}\beta_n (V_{inv} - V_{tn})^2 = \frac{1}{2}\beta_p (V_{inv} - V_{DD} - V_{tp})^2$$

where $\beta = K \frac{W}{L}$ and $K = \frac{\varepsilon_{ins}\varepsilon_0 \mu}{D}$. Solving for the logic threshold voltage V_{inv} , one gets

$$V_{inv} = \frac{V_{DD} + V_{tp} + V_{tn}\left(\frac{\beta_n}{\beta_p}\right)^{1/2}}{1 + \left(\frac{\beta_n}{\beta_p}\right)^{1/2}}.$$

Note that if $\beta_n = \beta_p$ and $V_{tn} = -V_{tp}$, then V_{inv}

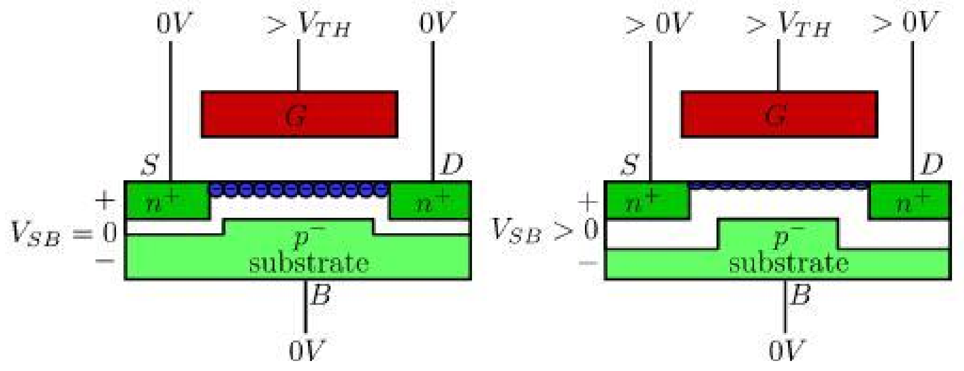
=0.5 V_{DD}

Region 4 : In this region, $V_{inv} < V_{in} \le V_{DD} - |V_{tp}|$. As the input voltage V_{in} is increased beyond V_{inv} ,
the *n* -transistor leaves saturation region and enters linear region, while the *p* -transistor continues
in saturation. The magnitude of both the drain current and the output voltage drops.

Region 5 : In this region, $V_{DD} - |V_{tp}| \le V_{in} \le V_{DD}$. At this point, the *p* -transistor is turned off, and the *n* -transistor is in linear region, drawing a small current, which falls to zero as V_{in} increases beyond V_{DD} -| V_{tp}|, since the *p* -transistor turns off the current path. The output in this region is $V_{out} \approx 0$.

BODY EFFECT:

Transistor is a 4-terminal device. Gate, drain and source are the 3 terminals that are used to control the transistor, but the bulk or body, if not properly biased, may put the transistor inoperable. **The pn junctions defined by source-bulk and drain-bulk**, which are basically two diodes, **must be reverse-biased** to stop them from leaking current from the source/drain to the substrate. That means that the source potential must always be equal or greater than the bulk potential. Since drain voltage is always greater or equal than source voltage, we don't even consider the drain-bulk junction.



When VS>VB, the depletion width of the pn junction increases That makes it more difficult to create a channel with the same VGS, effectively reducing the channel depth. In order to return to the same channel depth, VGS needs to increase accordingly.**The body effect can be seen as a change in threshold voltage**

**Channel
Length
modulation**
.

This in MOSFET is caused by the increase in depletion layer width at the drain as the drain voltage is increased. This leads to a shorter ΔL channel length (reduced by) and increased drain current. When the channel length of MOSFET is decreased and MOSFET is operated beyond ΔL channel pinch-off, the relative importance of pinchoff length with respect to physical length is increased.

Types of Design Rules

The design rules primary address two issues:

1. The geometrical reproduction of features that can be reproduced by the maskmaking and lithographical process ,and
2. The interaction between different layers.

There are primarily two approaches in describing the design rules.

1. Linear scaling is possible only over a limited range of dimensions.

2. Scalable design rules are conservative .This results in over dimensioned and less dense design.

3. This rule is not used in real life.

1. Scalable Design Rules (e.g. SCMOS, λ-based design rules):

In this approach, all rules are defined in terms of a single parameter **λ**. The rules are so chosen that a design can be easily ported over a cross section of industrial process ,making the layout portable .Scaling can be easily done by simply changing the value of.

The key disadvantages of this approach are:

2. **Absolute Design Rules (e.g. μ-based design rules) :**

In this approach, the design rules are expressed in absolute dimensions (e.g. 0.75μm) and therefore can exploit the features of a given process to a maximum degree. Here, scaling and porting is more demanding, and has to be performed either manually or using CAD tools .Also, these rules tend to be more complex especially for deep submicron.

The fundamental unity in the definition of a set of design rules is the minimum line width .It stands for the minimum mask dimension that can be safely transferred to the semiconductor material .Even for the same minimum dimension, design rules tend to differ from company to company, and from process to process. Now, CAD tools allow designs to migrate between compatible processes.

Layer Representations

With increase of complexity in the CMOS processes, the visualization of all the mask levels that are used in the actual fabrication process becomes inhibited. The layer concept translates these masks to a set of conceptual layout levels that are easier to visualize by the circuit designer. From the designer's viewpoint, all CMOS designs have the following entities:

• Two different substrates and/or wells: which are p-type for NMOS and n-type for

PMOS.

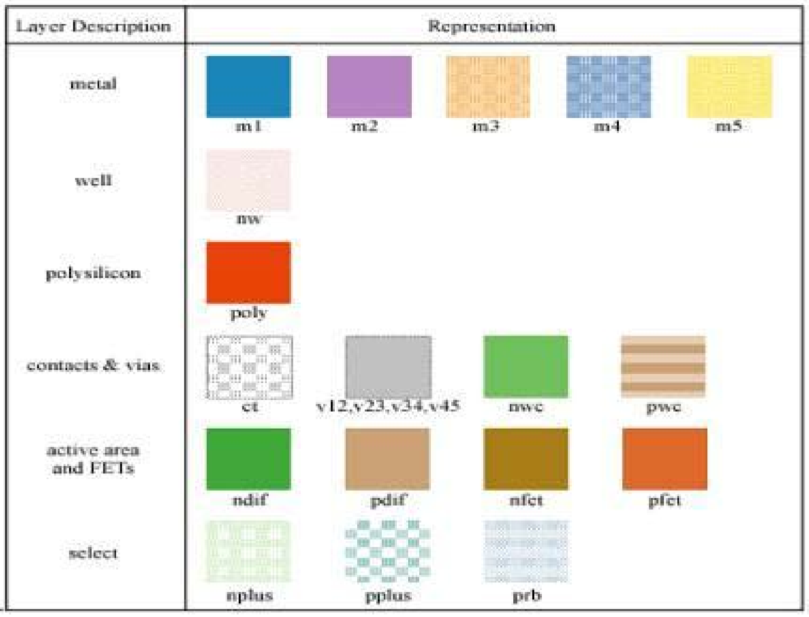
• Diffusion regions (p+ and n+): which defines the area where transistors can be formed. These regions are also called **active areas**. Diffusion of an inverse type is needed to implement contacts to the well or to substrate.These are called **select regions**.

• Transistor gate electrodes : Polysilicon layer

- Metal interconnect layers
- Interlayer contacts and via layers.

The layers for typical CMOS processes are represented in various figures in terms of:

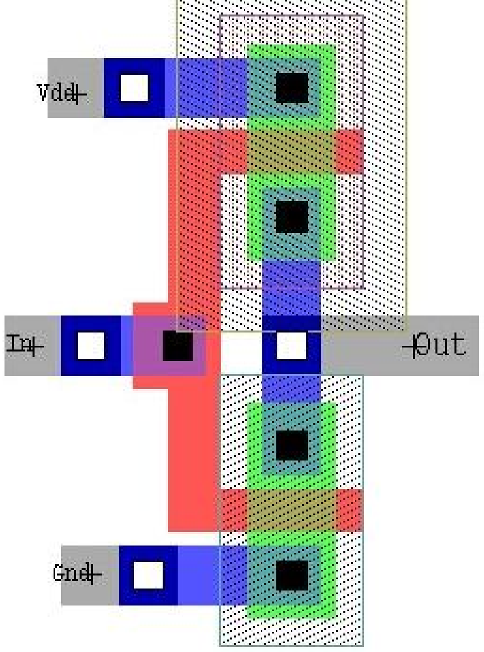
- A color scheme (Mead-Conway colors).
- Other color schemes designed to differentiate CMOS structures.
- Varying stipple patterns
- Varying line styles



Layer Description	Representation				
metal	🟦 m1	🟪 m2	▨ m3	▨ m4	🟨 m5
well	▨ nw				
polysilicon	🟥 poly				
contacts & vias	▨ ct	⬜ v12,v23,v34,v45	🟩 nwc	▨ pwc	
active area and FETs	🟩 ndif	🟫 pdif	🟫 nfet	🟧 pfet	
select	▨ nplus	▨ pplus	▨ prb		

Mead Conway Color coding for layers.

An example of layer representations for CMOS inverter using above design rules is shown below-



CMOS Inverter Layout Figure

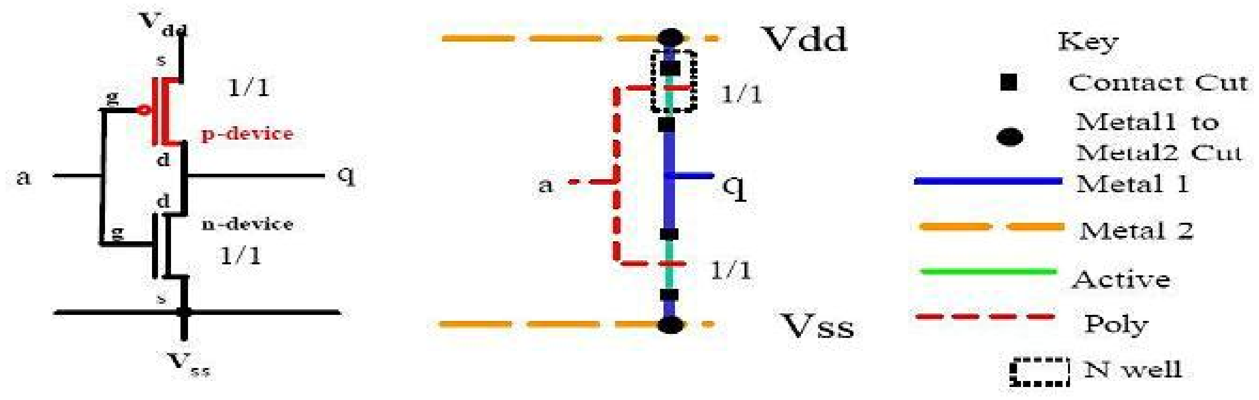
Stick Diagrams

Another popular method of symbolic design is **"Sticks"** layout. In this, the designer draws a freehand sketch of a layout, using colored lines to represent the various process layers such as diffusion, metal and polysilicon .Where polysilicon crosses diffusion, transistors are created and where metal wires join diffusion or polysilicon, contacts are formed.

This notation indicates only the relative positioning of the various design components. The absolute coordinates of these elements are determined automatically by the editor using a compactor. The compactor translates the design rules into a set of constraints on the component positions, and solve a constrained optimization problem that attempts to minimize the area or cost function.

The advantage of this symbolic approach is that the designer does not have to worry about design rules, because the compactor ensures that the final layout is physically correct. The disadvantage of the symbolic approach is that the outcome of the compaction phase is often unpredictable. The resulting layout can be less dense than what is obtained with the manual approach. In addition, it does not show exact placement, transistor sizes, wire lengths, wire widths, tub boundaries.

For example, stick diagram for CMOS Inverter is shown below.



Stick Diagram of a CMOS Inverter

LAYOUT DIAGRAM

Layout rules are used to prepare the photo mask used in the fabrication of integrated circuits. The rules provide the necessary communication link between the circuit designer and process engineer. Design rules represent the best possible compromise between performance and yield.

The design rules primarily address two issues -

1. The geometrical reproductions of features that can be reproduced by mask making and lithographical processes.

2. Interaction between different layers

Design rules can be specified by different approaches

1. **λ**-based design rules

2. **μ**-based design rules

As **λ**-based layout design rules were originally devised to simplify the industry-standard **μ**-based design rules and to allow scaling capability for various processes. It must be emphasized, however, that most of the submicron CMOS process design rules do not lend themselves to straightforward linear scaling. The use of **λ**-based design rules must therefore be handled with caution in sub-micron geometries.

λ-based Design Rules

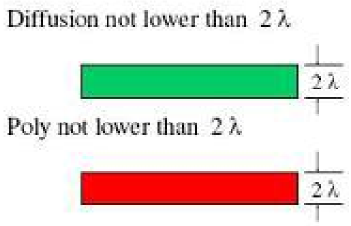
Features of λ-based Design Rules: λ-based Design Rules have the following features-

- **λ** is the size of a minimum feature
- All the dimensions are specified in integer multiple of **λ.**
- Specifying **λ** particularizes the scalable rules.

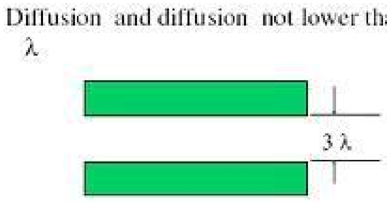
• Parasitic are generally not specified in **λ** units

• These rules specify geometry of masks, which will provide reasonable yields

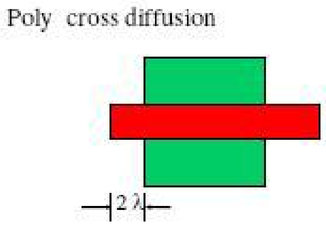
Guidelines for using λ-based Design Rules:



As, Minimum line width of poly is **2λ** & Minimum line width of diffusion is **2λ**

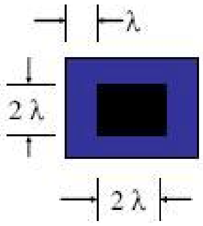


As Minimum distance between two diffusion layers **3λ**



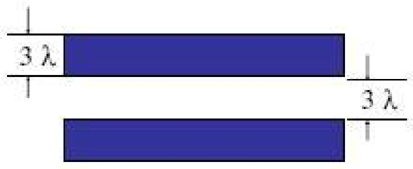
As It is necessary for the poly to completely cross active, other wise the transistor that has been created crossing of diffusion and poly, will be shorted by diffused path of source and drain.

Contact cut on metal



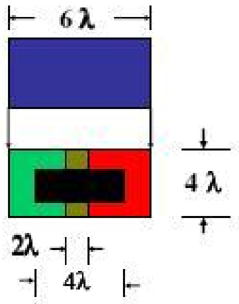
Contact window will be of **2λ** by **2λ** that is minimum feature size while metal deposition is of **4λ** by **4λ** for reliable contacts.

In Metal



Two metal wires have **3λ** distance between them to overcome capacitance coupling and high frequency coupling. Metal wires width can be as large as possible to decrease resistance.

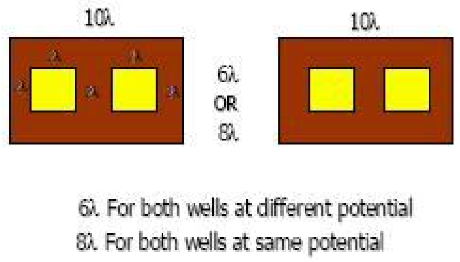
Buttering contact



Buttering contact is used to make poly and silicon contact. Window's original width is **4λ**, but on overlapping width is **2λ**.

So actual contact area is **6λ** by **4λ**.

The **distance between two wells** depends on the well potentials as shown above. The reason for 8l is that if both wells are at same high potential then the depletion region between them may touch each other causing punch-through. The reason for 6l is that if both wells are at different potentials then depletion region of one well will be smaller, so both depletion region will not touch each other so 6l will be good enough.


10λ
2λ
2λ
2λ
2λ
2λ
6λ
OR
8λ
10λ
6λ For both wells at different potential
8λ For both wells at same potential

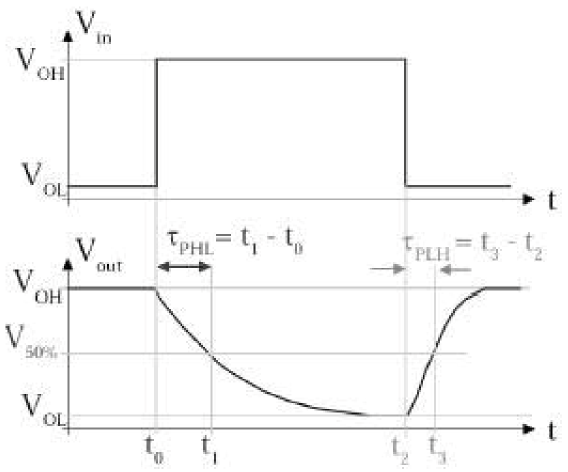
The active region has length **10λ** which is distributed over the followings-

- **2λ** for source diffusion
- **2λ** for drain diffusion
- **2λ** for channel length
- **2λ** for source side encroachment
- **2λ** for drain side encroachment

Basic Definitions in Delay:

Before calculating the propagation delay of CMOS Inverter, we will define some basic terms-

- **Switching speed -** limited by time taken to charge and discharge, **C**L.
- **Rise time, tr:** waveform to rise from 10% to 90% of its steady state value
- **Fall time tf:** 90% to 10% of steady state value
- **Delay time, t*d*:** time difference between input transition (50%) and 50% output level



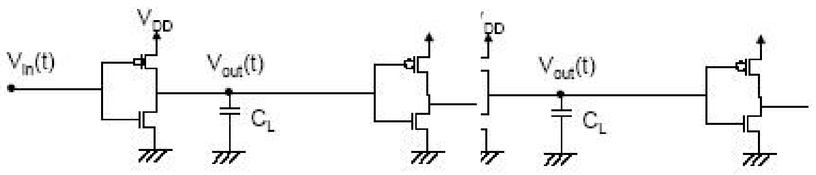
Propagation delay graph

The propagation delay **tp** of a gate defines how quickly it responds to a change at its inputs, it expresses the delay experienced by a signal when passing through a gate. It is measured between the 50% transition points of the input and output waveforms as shown in the figure 16.1 for an inverting gate. The τ_{pHL} defines the response time of the gate for a low to high output transition, while τ_{pHL} refers to a high to low transition. The propagation delay τ_p as the average of the two

$$\tau_p = (\tau_{pLH} + \tau_{pHL})/2$$

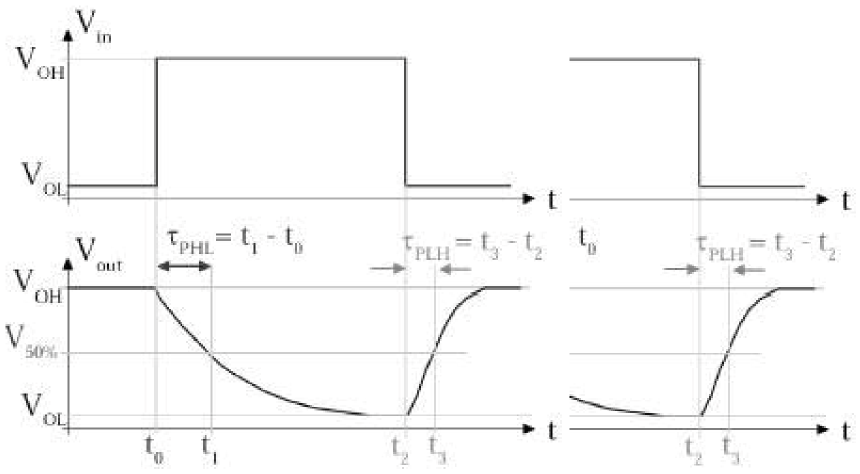
Quick Estimates:

We will give an example of how to calculate quick estimate. From fig, we can write following equations.



Example CMOS Inverter Circuit

$$V_{50\%} = V_{OL} + \frac{V_{OH} - V_{OL}}{2} = \frac{V_{OH} + V_{OL}}{2}$$
$$V_{90\%} = V_{OL} + 0.9(V_{OH} - V_{OL})$$
$$V_{10\%} = V_{OL} + 0.1(V_{OH} - V_{OL})$$



Propagation Delay of above MOS circuit

From figure, when Vin = 0 the capacitor CL charges through the PMOS, and when

Vin = 5 the capacitor discharges through the N-MOS. The capacitor current is –

$$C_L \frac{dV}{dt} = i_{dsn} = | i_{dsp} |$$

From this the delay times can be derived as

$$\int dt = \int \frac{C_L}{i_{ds}} dV$$

The expressions for the propagation delays as denoted in the figure can be easily seen to be

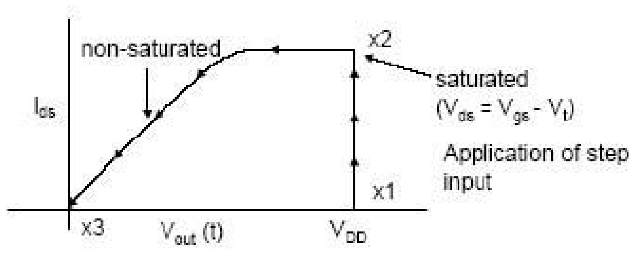
$$\tau_{PHL} = \frac{C_{Load}\Delta V_{HL}}{I_{avg,HL}} = \frac{C_{Load}(V_{OH} - V_{50\%})}{I_{avg,HL}} \quad \& \quad \tau_{PLH} = \frac{C_{Load}\Delta V_{LH}}{I_{avg,LH}} = \frac{C_{Load}(V_{50\%} - V_{OL})}{I_{avg,HL}}$$

where $I_{avg,HL}$ & $I_{avg,LH}$ are defined as -

$$I_{avg,HL} = \frac{1}{2}[i_C(V_{in} = V_{OH}, V_{out} = V_{OH}) + i_C(V_{in} = V_{OH}, V_{out} = V_{50\%})]$$

$$I_{avg,LH} = \frac{1}{2}[i_C(V_{in} = V_{OL}, V_{out} = V_{OL}) + i_C(V_{in} = V_{OL}, V_{out} = V_{50\%})]$$

Rise and Fall Times



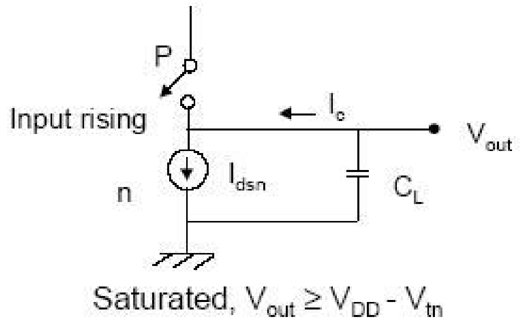
trjectory of n-transistor operating point

Above Figure shows the trajectory of the n-transistor operating point as the input voltage, **Vin(t)**, changes from **0V** to **VDD**. Initially, the end-device is cutt-off and the load capacitor is charged to **VDD**. This illustrated by **X1** on the characteristic curve. Application of a step voltage (**VGS = VDD**) at the input of the inverter changes the operating point to **X2**. From there onwards the trajectory moves on the **VGS= VDD** characteristic curve towards point **X3** at the origin.

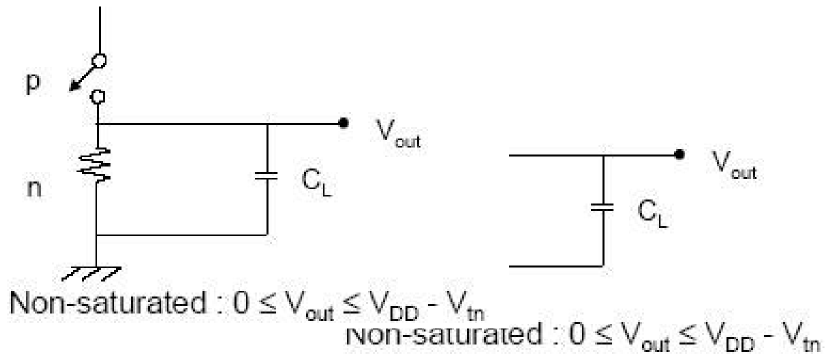
Thus it is evident that the fall time consists of two intervals:

1. **t**f1=period during which the capacitor voltage, **Vout**, drops from **0.9VDD** to (**VDD–Vtn**)

2. tf2=period during which the capacitor voltage, **Vout**, drops from **(VDD–Vtn)** to **0.1VDD.**



Equivalent circuit for showing behav. of **t**f1


p
V_{out}
n
C_L
Non-saturated : $0 \leq V_{out} \leq V_{DD} - V_{tn}$
V_{out}
C_L
Non-saturated : $0 \leq V_{out} \leq V_{DD} - V_{tn}$

Equivalent circuit for showing behav. of **t**f2

As we saw in last section, the delay periods can be derived using the general equation

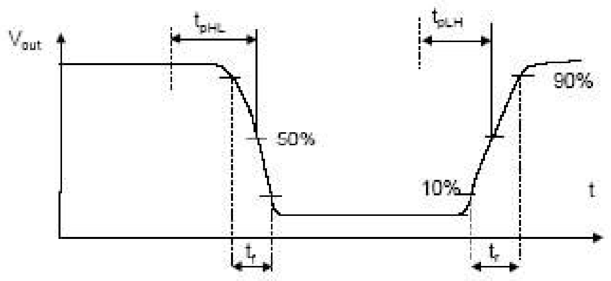
$$\int dt = \int \frac{C_L}{i_{ds}} dV$$

while in saturation,

$$I_{dsn(sat)} = \frac{\beta_n}{2}(V_{in} - V_{tn})^2$$

Integrating from t = t**1**, corresponding to **Vout=0.9 VDD**, to t = t**2** corresponding to **V**out=**(*V*DD-*V*tn) results in,**

$$t_{f1} = \frac{2C_L}{\beta_n(V_{DD} - V_{tn})^2} \int_{V_{DD}-V_{tn}}^{0.9V_{DD}} dV_{out}$$
$$= \frac{2C_L(V_{DD} - 0.1V_{tn})}{\beta_n(V_{DD} - V_{tn})^2}$$



Rise and Fall time graph

When the n-device begins to operate in the linear region, the discharge current is no longer constant. The time **tf1** taken to discharge the capacitor voltage from **(*V*DD-*V*tn)** to ***0.1V*DD** can be obtained as before. In linear region,

$$I_{dsn(\text{linear})} = -\beta_n [(V_{DD} - V_{tn})V_{out} - V_{out}^2/2]$$

$$t_{f2} = \frac{C_L}{\beta_n (V_{DD} - V_{tn})^2} \int_{V_{DD}-V_{tn}}^{0.1V_{DD}} \frac{dV_{out}}{\frac{V_{out}^2}{2(V_{DD} - V_{tn})} - V_{out}} = \frac{C_L}{\beta_n (V_{DD} - V_{tn})} \ln\left(\frac{19V_{DD} - 20V_{tn}}{V_{DD}}\right)$$

$$= \frac{C_L}{\beta_n V_{DD}(1-n)} \ln(19 - 20n) \qquad \text{where } n = \frac{V_{tn}}{V_{DD}}$$

Thus the complete term for the fall time is,

$$t_f = t_{f1} + t_{f2} = \frac{2C_L}{\beta_n V_{DD}(1-n)}\left[\frac{(n-0.1)}{(1-n)} + \frac{1}{2}\ln(19-20n)\right]$$

The fall time tf can be approximated as,

$$t_f \approx k_n \frac{C_L}{\beta_n V_{DD}} \qquad k_n = 3 \sim 4 \text{ for } V_{DD} = 3 \sim 5V \text{ and } V_{tn} = 0.5 \sim 1V$$

From this expression we can see that the delay is directly proportional to the load capacitance. Thus to achieve high speed circuits one has to minimize the load capacitance seen by a gate. Secondly it is inversely proportion to the supply voltage i.e. as the supply voltage is raised the delay time is reduced. Finally, the delay is proportional to the **βn** of the driving transistor so increasing the width of a transistor decreases the delay.

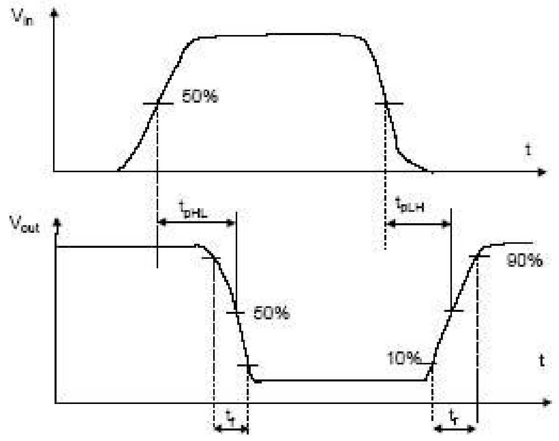
Due to the symmetry of the CMOS circuit the rise time can be similarly obtained as; For equally sized **n** and **p** transistors (where **βn=2βp**) **tf**=t**r**

Thus the fall time is faster than the rise time primarily due to different carrier mobilites associated with the p and n devices thus if we want t**f**=t**r** we need to make **βn**/**βp** =**1**. This implies that the channel width for the **p**-device must be increased to approximately 2 to 3 times that of the **n**-device.

The propagation delays if calculated as indicated before turn out to be,

$$\tau_{PLH} = \frac{C_L}{k_p(V_{DD} - |V_{T0p}|)}\left[\frac{2|V_{T0p}|}{(V_{DD} - |V_{T0p}|)} + \ln\left(\frac{4((V_{DD} - |V_{T0p}|))}{V_{DD}} - 1\right)\right]$$

$$\tau_{PHL} = \frac{C_L}{k_n(V_{DD} - V_{T0n})}\left[\frac{2V_{T0n}}{(V_{DD} - V_{T0n})} + \ln\left(\frac{4((V_{DD} - V_{T0n}))}{V_{DD}} - 1\right)\right]$$



Rise and Fall time graph of Output w.r.t Input

If we consider the rise time and fall time of the input signal as well, then

$$\tau_{PLH(\text{actual})} = \sqrt{(\tau_{PLH})^2 + (t_f/2)^2}$$
$$\tau_{PHL(\text{actual})} = \sqrt{(\tau_{PHL})^2 + (t_r/2)^2}$$

These are the rms values for the propagation delays.

SCALING OF MOS TRANSISTOR:

Types of Scaling

Two types of scaling are common:

1) constant field scaling and
2) constant voltage scaling.

Constant field scaling yields the largest reduction in the power-delay product of a single transistor. However, it requires a reduction in the power supply voltage as one decreases the minimum feature size.

Constant voltage scaling does not have this problem and is therefore the preferred scaling method since it provides voltage compatibility with older circuit technologies. The disadvantage of constant voltage scaling is that the electric field increases as the minimum feature length is reduced. This leads to velocity saturation, mobility degradation, increased leakage currents and lower breakdown voltages. After scaling, the different Mosfet parameters will be converted as given by table below:

Before Scaling After Constant Field Scaling After Constant Voltage Scaling

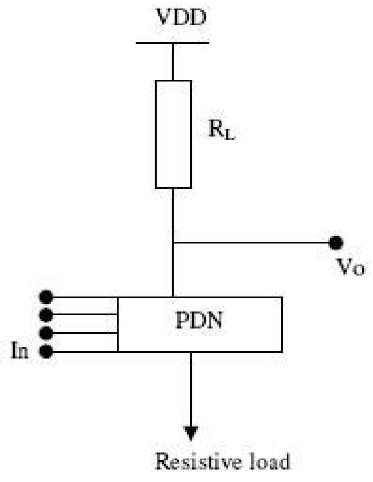
Before Scaling	*After Constant Field Scaling*	*After Constant Voltage Scaling*
L	$L' = L/s$	$L' = L/s$
W	$W' = W/s$	$W' = W/s$
t	$t'_{ox} = t_{ox}/s$	$t'_{ox} = t_{ox}/s$
x_i	$x'_i = x_i/s$	$x'_i = x_i/s$
V_{DD}	$V'_{DD} = V_{DD}/s$	$V'_{DD} = V_{DD}$
V_{Th}	$V'_{Th} = V_{Th}/s$	$V'_{Th} = V_{Th}$
N_a or N_d	$N'_a = N_a * s$ *or* $N'_d = N_d * s$	$N'_a = N_a * s^2$ *or* $N'_d = N_d * s^2$
C_{ox}	$C'_{ox} = C_{ox} * s$	$C'_{ox} = C_{ox} * s$
I_{DS}	$I'_{DS} = I_{DS}/s$	$I'_{DS} = I_{DS} * s$
P_D	$P'_D = P_D/s^2$	$P'_D = P_D * s$

UNIT III

SUBSYSTEM DESIGN & LAYOUT

Ratioed Logic:

Instead of combination of active pull down and pull up networks such a gate consists of an NMOS pull down network that realizes the logic function and a simple load device. For an inverter PDN is single NMOS transistor.



Ratioed Logic Circuit

The load can be a passive device, such as a resistor or an active element as a transistor. Let us assume that both PDN and load can be represented as linearized resistors. The operation is as follows: For a low input signal the pull down network is off and the output is high by the load. When the input goes high the driver transistor turns on, and the resulting output voltage is determined by the resistive division between the impedances of pull down and load network:

VOL= RDVDD/(RD+RL)

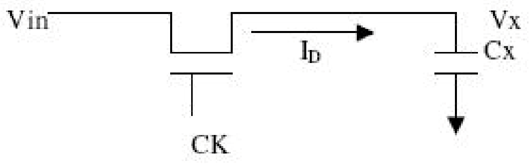
where **R**D = pulldown n/w resistance, **R**L= load resistance.

To keep the low noise margin high it is important to chose **RL>>RD**. This style of logic therefore called ratioed, because a careful **PDN** scaling of impedances (or transistor sizes) is required to obtain a workable gate. This is in contrast to the ratioless logic style as complementary CMOS, where the low and high level don't depend upon transistor sizes. As a satisfactory level we keep **RL>=4RD**. To achieve this, **(W/L)D/(W/L)L> 4.**

Pass Transistor Logic

The fundamental building block of nMOS dynamic logic circuit, consisting of an nMOS

pass transistor is shown in figure



Pass Transistor Logic Circuit

The pass transistor MP is driven by the periodic clock signal and acts as an access switch to either charge up or down the parasitic capacitance, **C**x, depending on the input signal **V**in. Thus there are 2 possible operations when the clock signal is active are the logic "**1**" transfer(charging up the capacitance **C**x to logic high level) and the logic "**0**" transfer(charging down the capacitance **C**x to a logic low level). In either case, the output of the depletion load of the nMOS inverter obviously assumes a logic low or high level, depending on the voltage **V**x. The pass transistor MP provides the only current path to the intermediate capacitive node X. when clock signal becomes inactive **(clk=0)** the pass transistor ceases to conduct and the charge is stored in the parasitic capacitor **C**x continues to determine the output level of the inverter.

Logic “1” Transfer: Assume that the **V**x = **0** initially. A logic "**1**"level is applied to the input terminal which corresponds to **V**in=**V**OH=**V**DD. Now the clock signal at the gate of the pass transistor goes from **0** to **V**DD at **t=0**. It can be seen that the pass transistor starts to conduct and operate in saturation throughout this cycle since VDS=VGS. Consequently **V**DS> **V**GS**V**tn.

Analysis: The pass transistor operating in saturation region starts to charge up the capacitor **C**x, thus:

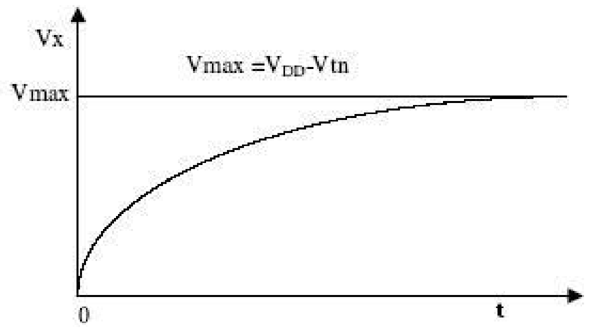
$$C_X (dV_X/dt) = (k_n/2)(V_{DD} - V_X - V_{tn})^2 \Rightarrow \int_0^t dt = (2C_X/k_n) \int_0^{V_X} dV_X/(V_{DD} - V_X - V_{tn})^2$$

So. $$t = (2C_X/k_n)[1/(V_{DD} - V_X - V_{tn}) - 1/(V_{DD} - V_{tn})]$$

The previous equation for **Vx(t)** can be solved as-

$$V_X(t) = (V_{DD} - V_{tn}) \frac{(k_n/2C_X)(V_{DD} - V_{tn})t}{1+(k_n/2C_X)(V_{DD} - V_{tn})t}$$

The variation of the node voltage **Vx(t)**is plotted as a function of time in fig. The voltage rises from its initial value of **0** and reaches **Vmax =VDD-Vtn** after a large time. The pass transistor will turn off when **V**x = **V**max. Since **Vgs= Vtn**. Therefore **Vx** can never attain **VDD** during logic **1** transfer. Thus we can use buffering to overcome this problem.



Node Voltage Vx vs t

Logic “0” Transfer: Assume that the ***V*x=*1***

Initially. A logic“**0**” level is applied to the input terminal which corresponds to ***V*in=*1***. Now the clock signal at the gate of the pass transistor goes from **0** to **VDD** at **t=0**. It can be seen that the pass transistor starts to conduct and operate in linear mode throughout this cycle and the drain current flows in the opposite direction to that of charge up.

Analysis: We can write –

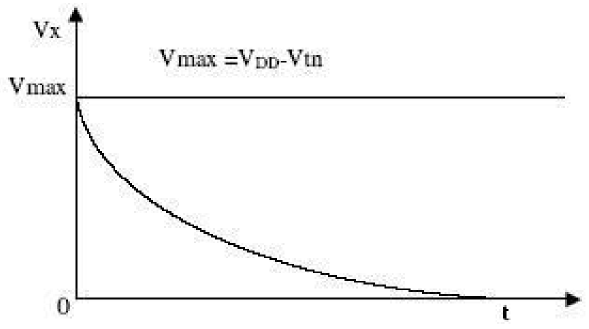
$$-C_X(dV_X/dt) = (k_n/2)[(V_{DD} - V_{tn})V_X - V_X^2] \Rightarrow \int_0^t dt = -(C_X/k_n)\int_0^{V_X} dV_X/[(V_{DD} - V_{tn})V_X - V_X^2]$$

So, $t = (C_X/k_n)\ln[2((V_{DD} - V_{tn}) - V_X)/V_X]$

The above equation for **Vx(t)** can be solved as –

$$V_X(t) = \frac{2(V_{DD} - V_{tn})}{1 + e^{(k_n/C_X)t}}$$

Plot of **Vx(t)** is shown in figure



Node Voltage Vx vs t

Dynamic Logic Circuits

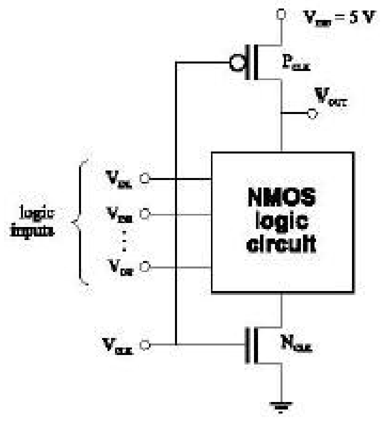
In case of static CMOS for a fan-in of **N**, **2N** transistors are required. In order to reduce this, various other design logics were used like pseudo-NMOS logic and pass transistor logic. However the static power consumption in these cases increased. An alternative to these design logics is **Dynamic logic**, which reduces the number of transistors at the same time keeps a check on the static power consumption.

Principle: A block diagram of a dynamic logic circuit is as shown in fig 19.31. This uses

NMOS block to implement its logic

The operation of this circuit can be explained in two modes.

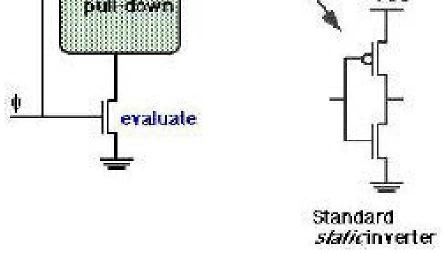
1. Precharge
2. Evaluation



Dynamic CMOS Block Diagram

In the precharge mode, the **CLK** input is at logic **0**. This forces the output to logic **1**, charging the load capacitance to **VDD**. Since the NMOS transistor **M1** is off the pulldown path is disabled. There is no static consumption in this case as there is no direct path between supply and ground.

In the evaluation mode, the **CLK** input is at logic **1**. Now the output depends on the PDN block. If there exists a path through PDN to ground (i.e. the PDN network is **ON**), the capacitor **CL** will discharge else it remains at logic **1**.As there exists only one path between the output node and a supply rail, which can only be ground, the load capacitor can discharge only once and if this happens, it cannot charge until the next precharge operation. Hence the inputs to the gate can make at most one transition during evaluation



DOMINO CMOS Block Diagram

Advantages of dynamic logic circuits:

1. As can be seen, the number of transistors required here are **N+2** as compared to

2N in the Static CMOS circuits.

2. This circuit is still a ratioless circuit as in Static case. Hence, progressive sizing and ordering of the transistors in the PDN block is important.

3. As can be seen, the static power loss is negligible.

Disadvantages of dynamic logic circuits:

1. The penalty paid in such circuits is that the clock must run everywhere to each such block as shown in the diagram.

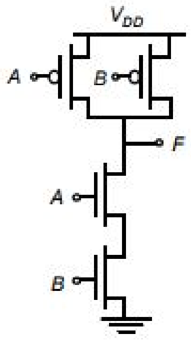
2. The major problem in such circuits is that the output node is at Vdd till the end of the precharge mode. Now if the **CLK** in the next block arrives earlier compared to the **CLK** in this block, or the PDN network in this block takes a longer time to evaluate its output, then the next block will start to evaluate using this erroneous value

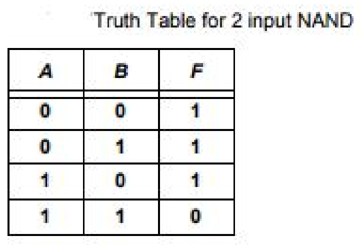
The second part of the disadvantage can be eliminated by using **DOMINO CMOS** circuits which are as shown below.

As can be seen the output at the end of precharge is inverted by the inverter to logic 0. Thus the next block will not be evaluated till this output has been evaluated. As an ending point, it must be noted that this also has a disadvantage that since at each stage the output is inverted, the logic must be changed to accommodate this.

STATIC CMOS LOGIC:

The most widely used logic style is static complementary CMOS. The static CMOS style is really an extension of the static CMOS inverter to multiple inputs. In review, the primary advantage of the CMOS structure is robustness (i.e, low sensitivity to noise), good performance, and low power consumption (with no static power consumption). As we will The complementary CMOS circuit style falls under a broad class of logic circuits called static circuits in which at every point in time (except during the switching transients), each gate output is connected to either VDD or Vss via a low-resistance path. Also, the outputs of the gates assume at all times the value of the Boolean function implemented by the circuit (ignoring, once again, the transient effects during switching periods). This is in contrast to the dynamic circuit class, that relies on temporary storage of signal values on the capacitance of high-impedance circuit nodes. The latter approach has the advantage that the resulting gate is simpler and faster. On the other hand, its design and operation are more involved than those of its static counterpart, due to an increased sensitivity to noise.




Truth Table for 2 input NAND

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

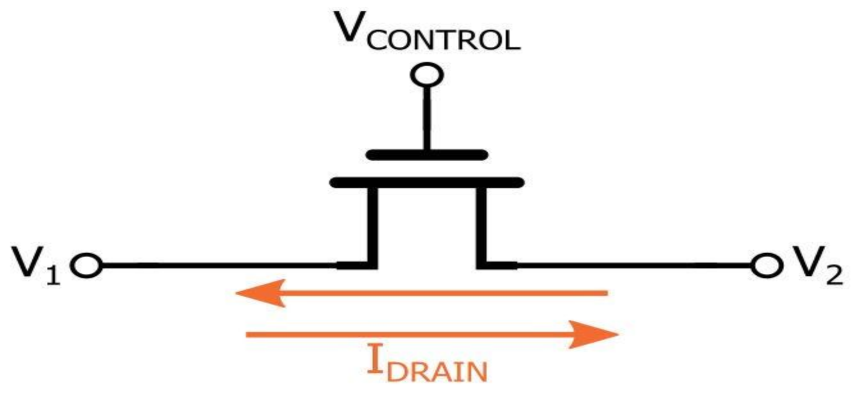
Two I/P NAND gate in complementary Static CMOS Style

CMOS TRANSMISSION GATE:

We usually see MOSFETs arranged with their sources and drains connected—either directly or through, for example, a resistor or active load—to positive and negative supply rails, with the gate acting as the input terminal. This is true in both analog circuits, such as the common-source amplifier, and digital circuits, such as the ubiquitous CMOS inverter. It’s good to remember, though, that the MOSFET is not limited to configurations such as these.

The channel created by a sufficiently high gate-to-source voltage allows current to flow between the source and drain terminals, and in this sense the MOSFET is a voltage-controlled switch. Thus, there is no law that prevents us from using the source and drain as input and output terminals, with the control voltage applied to the gate.

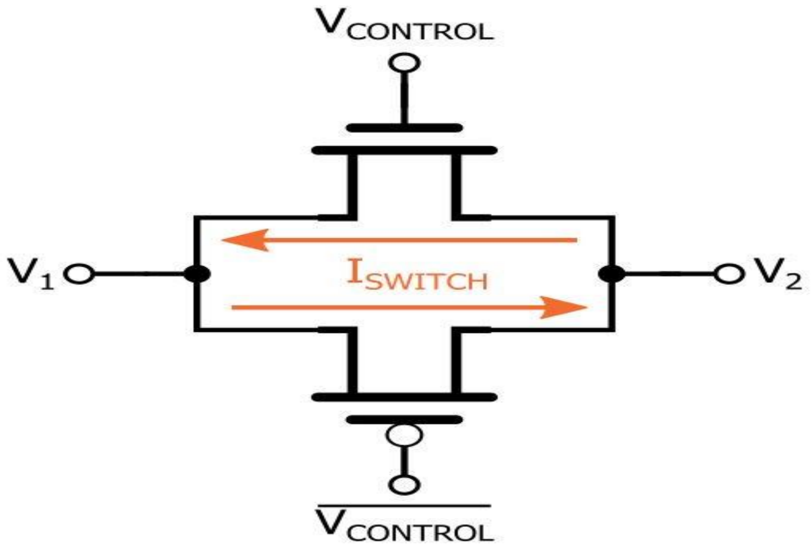
A single NMOS (or PMOS) transistor can be used as a voltage-controlled switch. The “circuit” (really just a single transistor) is the following:



the arrow that usually identifies the source is removed. This is because the source terminal actually changes according to whether V_1 is higher than V_2 or V_2 is higher than V_1. Also, the use of V_1 and V_2 instead of V_{IN} and V_{OUT} is intended to emphasize that this single NMOS transistor can indeed conduct current in both directions.

As probably expected, this circuit is far from a perfect switch. One problem is the source voltage: The current through the MOSFET is influenced by the source voltage, and the source voltage depends on whatever signal is passing through the switch. Indeed, if the gate is controlled by a driver that cannot exceed V_{DD}, the transistor can pass signals only as high as V_{DD} minus the threshold voltage. This threshold-voltage limitation is made even worse by the body effect, which comes into play when the FET’s source and body terminals are not at the same potential.

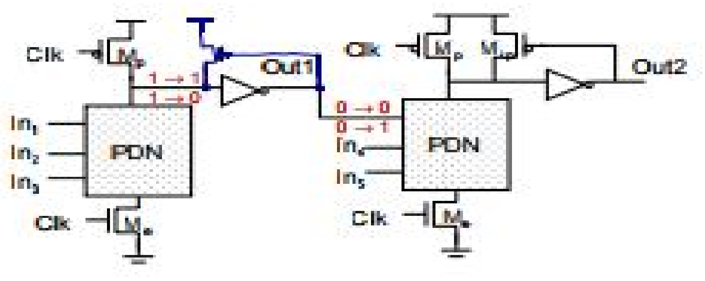
When you analyze and ponder this switch, you recognize a certain asymmetry. For example, if we are using this switch for pass-transistor logic, the NMOS can effectively pass a logic-low signal but not a full logic-high signal. Is it possible to modify the circuit in a way that will redress this asymmetry? If you are maintaining a good CMOS mentality, your intuition might tell you that we could achieve better overall performance by incorporating a PMOS transistor to compensate for the deficiencies of the NMOS.



Here we have a PMOS in parallel with the NMOS; I used an “invert” circle to identify the PMOS transistor. Note that the control signal applied to the PMOS is the complement of the control signal applied to the NMOS; this is reminiscent of the CMOS inverter, where a logic-high voltage turns on the NMOS and a logic-low voltage turns on the PMOS.

This CMOS transmission gate is a synergistic system—the NMOS provides good switch performance under conditions that are favorable for itself but not for the PMOS, and the PMOS provides good switch performance under conditions that are favorable for itself but not for the NMOS. The result is a simple yet effective bidirectional voltage-controlled switch that is suitable for both analog and digital applications.

DOMINO LOGIC:

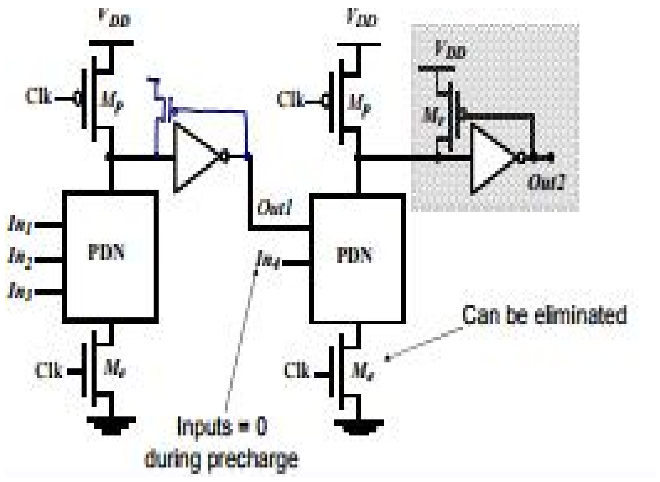


Properties of Domino Logic

- Only non-inverting logic can be implemented ‰
- Very high speed ƒ

- static inverter can be skewed, only L-H transition critical *f*
- Input capacitance reduced – smaller logical effort

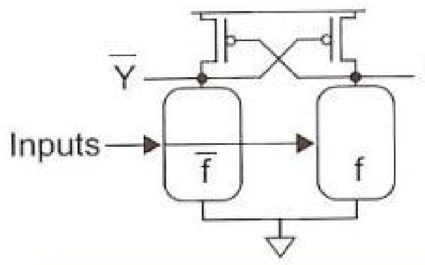
DESIGNING WITH DOMINO LOGIC:



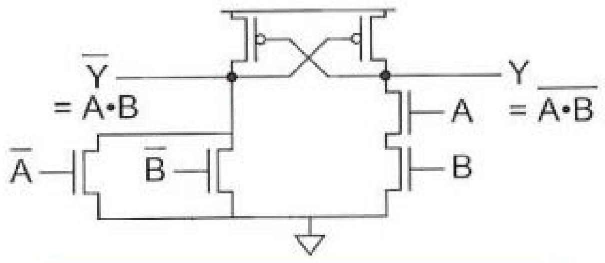
DIFFERENTIAL CASCODE VOLTAGE SWITCH LOGIC:

- Performance advantage of ratioed circuits without the extra power

• Requires complementary inputs – produces complementary outputs
• Operation – two nMOS arrays one for f, one for f – cross-coupled load pMOS – one path is always active
 • since either f or f is always true – other path is turned off
 • no static power generic differential logic gate differential AND/NAND gate (logic arrays turns off one load)



generic differential logic gate



differential AND/NAND gate

Advantages of CVSL :

low load capacitance on inputs
no static power consumption
automatic complementary functions

Disadvantages:

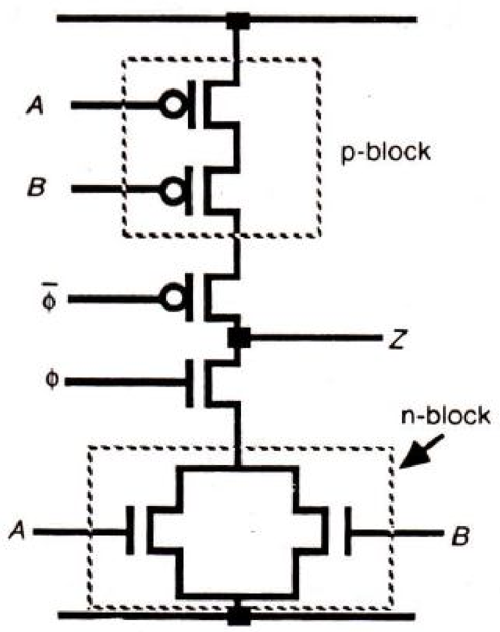
requires complementary inputs

more transistors for single function

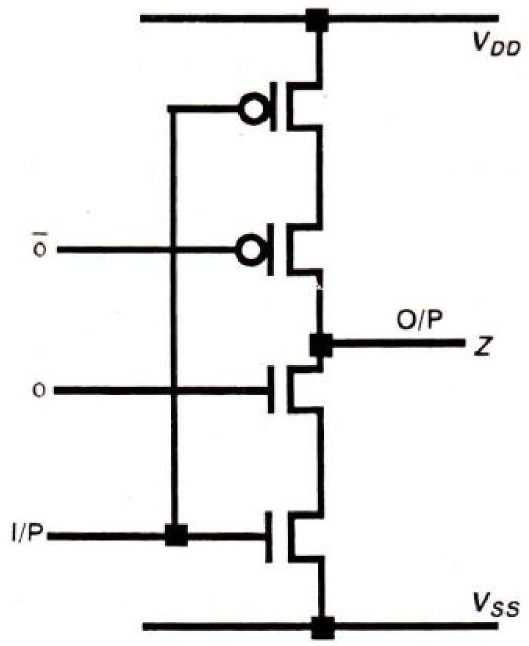
Clocked CMOS **logic**

The general arrangement may be made clearer by. The logic is implemented in both n- and p-transistors in the form of a pull-up p-block and a complementary n-block pulldown structure as for the inverter-based CMOS logic discussed earlier.

However, the logic in this case is evaluated (connected to the output) only during the on period of the clock. As might be expected, a clocked inverter circuit forms part of this family of logic as shown in Figure. Owing to the extra transistors in series with the output,slower rise-times and fall -times ca.n be expected



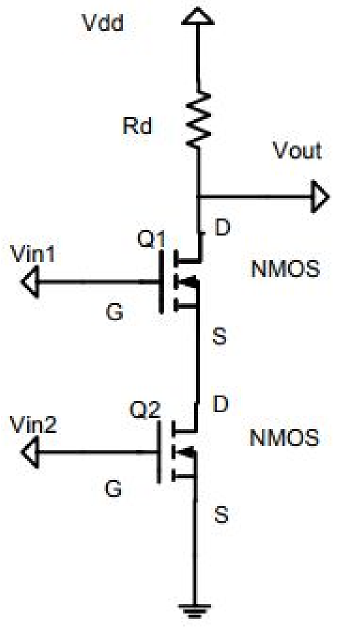
(a) 2 1/P *Nor* gate



(b) Inverter

NMOS NAND Gate:

NMOS NAND Gate Use Vdd = 9.0Vdc. For the NMOS NAND gate shown below gate, using the 2N7000 MOSFET LTspice model such that Vto = 2.0. The input logic “1” = 9 volt and ground as a logic “0”. Make a truth table showing the four possible combinations of Vin1 and Vin2 and the outputs. Choose Rd (drain current limit resistor) such that the drain currents of the NMOS devices will be about 30mA when the Vout is in a low state. Then run a DC Bias Point simulation (use the added 2N7000 model in LTspice) on your design with the four possible input combinations for Vin1 and Vin2 to verify your gate. Observe the output voltage value for each input combination. Print your circuit schematic showing voltages for all four input combination

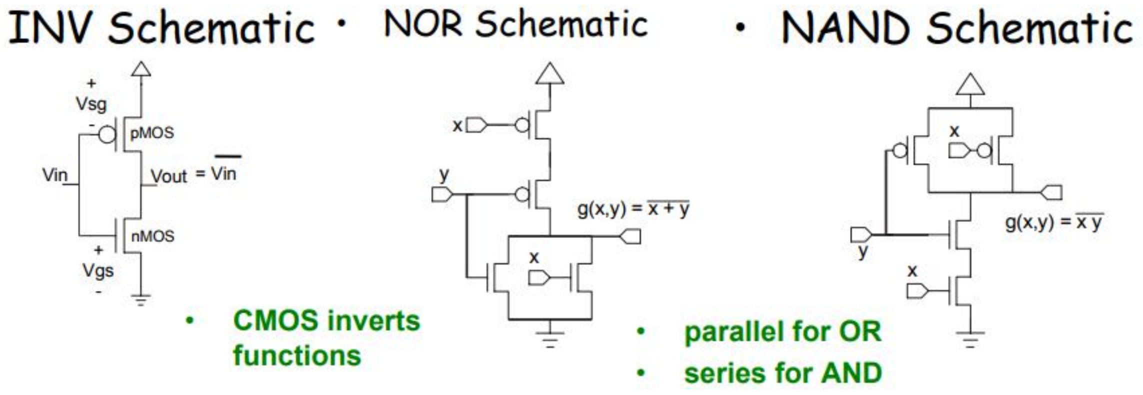


Nmos NAND GATE

NMOS NOR Gate:

NMOS NOR Gate Use Vdd = 9.0Vdc. Design an NMOS NOR gate using the 2N7000 MOSFET the model has Vto = 2.0 . Limit the drain current total to 30mA with a drain resistor (Rd). Show all work for your design and drawing. Then simulate your design in LTspice with DC Bias Point simulations as you did for the NAND gate. Print out your circuit schematic showing voltages for all four input combination add from the view menu node voltage and drian current to display on the schematic. Also, fill in the truth table with all of the Bias Point simulation voltage values

CMOS Inverter ,NAND and NOR gates:



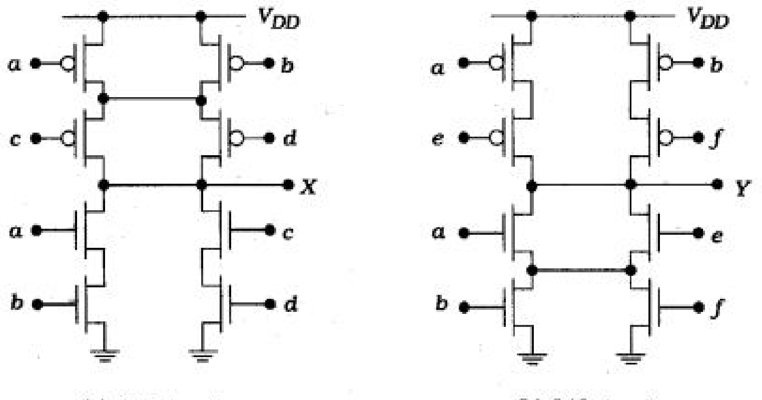
CMOS COMBINATIONAL LOGIC:

Use DeMorgan relations to reduce functions

• remove all NAND/NOR operations

• implement nMOS network

• create pMOS by complementing operations



(a) AOI circuit (b) OAI circuit

Multiplexers (Data Selectors)

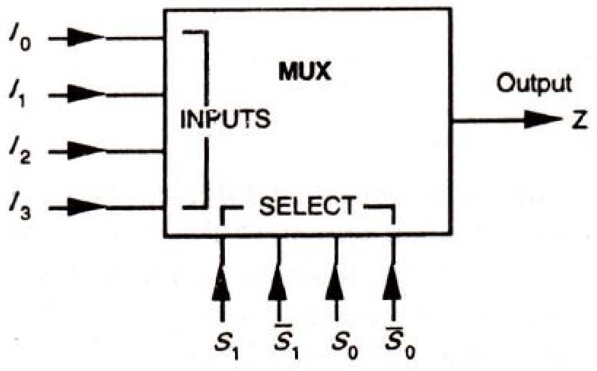
Multiplexers are widely used and have many applications. They are also commonly available in a number of standard configurations in TTL and other logic families. In order to arrive at a standard cell for multiplexers, we will consider a commonly used circuit, the four-way multiplexer.

The requirements and general arrangement of a four-way multiplexer are set out in Figure from which we may write

$$\boxed{Z = I_0.\overline{S}_1.\overline{S}_0 + I_1.\overline{S}_1.S_0 + I_2.S_1.\overline{S}_0 + I_3.S_1.S_0}$$

where S1 and S0 are the selector inputs. Note that in this case we do not need to be concerned about undefined ouput conditions since, if S1 and S0 have defined logic states, output *Z* must always be connected to one of I_0 to I_3.

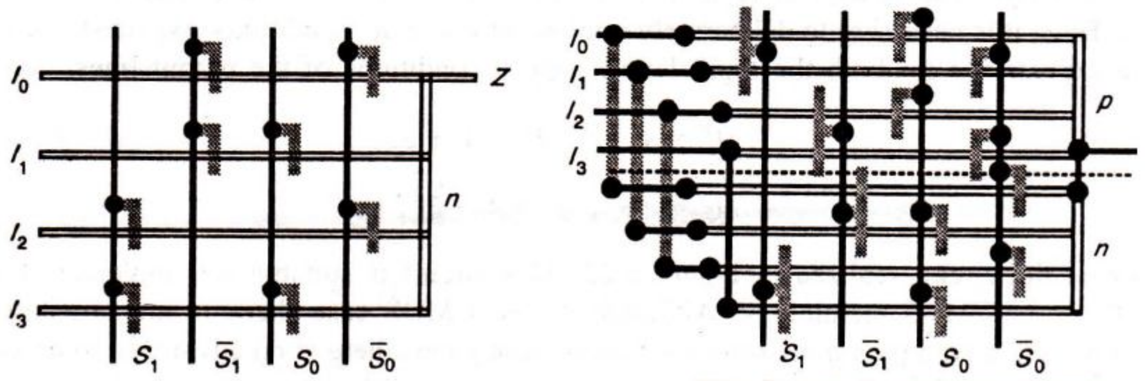
A transmission-gate-based CMOS Stick diagram is given



Truth table

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

Selector logic circuit.



Note: V_{DD} and V_{SS} contacts not shown.

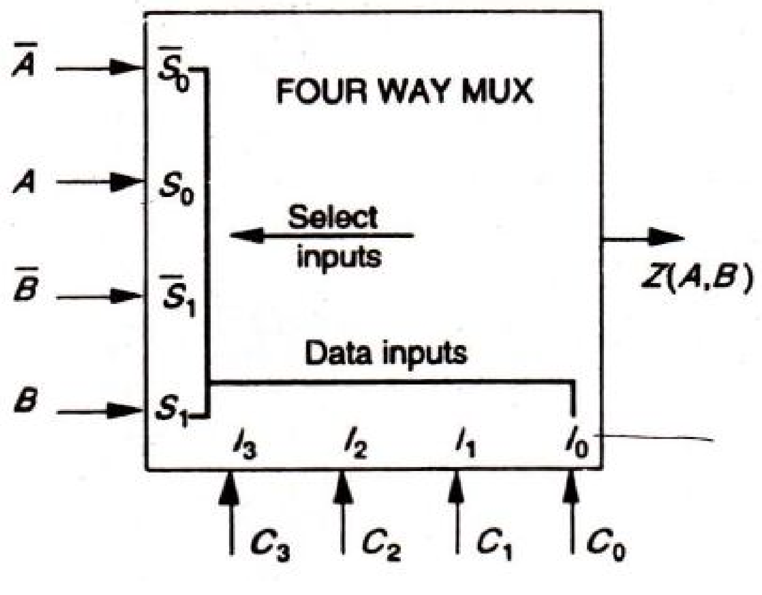
(a) nMOS switches (b) Transmission gates (CMOS)

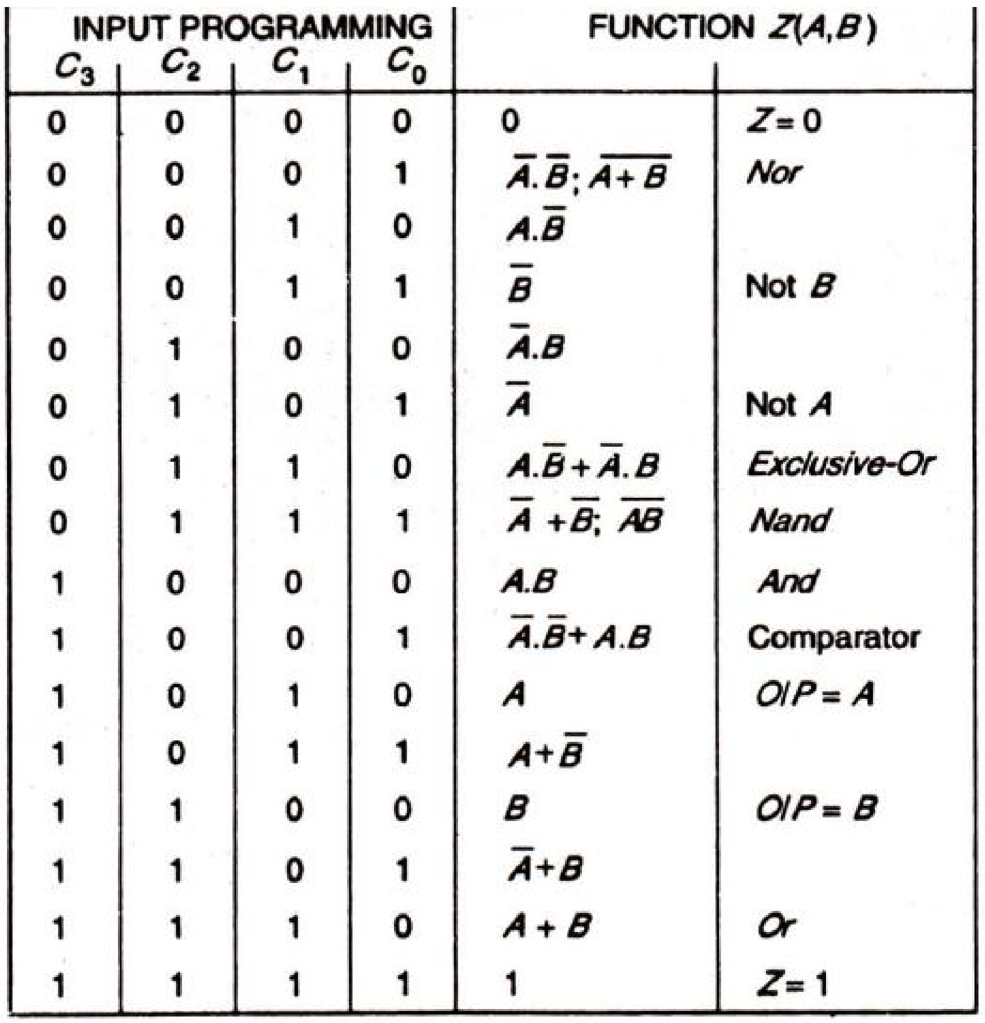
Switch logic implementations of a four-way multiplexer.

For the nMOS case a standard cell is illustrated in Figure . The standard cell in this case measured 7 λ. x 11 λ and is shown in the dotted outline.Two versions of the cell are needed to complete the network, one version with a pass transistor as shown and the other version without. If computer-aided design tools are used, the two versions may be designed as one cell suitably parameterized to include or exclude the pass transistor.

A General Logic Function Block

An arrangement to generate any function of two variables (A, *B)* is readily formed from any form of four-way multiplexer. It will be seen that the required function is generated by driving the multiplexer select inputs from the required two variables*A* and *B* and by 'programming' the inputs I_0 to I_3 appropriately with Os and 1 s, as indicated in the figure. Larger multiplexers may be similarly employed to generate any function of up to four variables (16-way multiplexer).





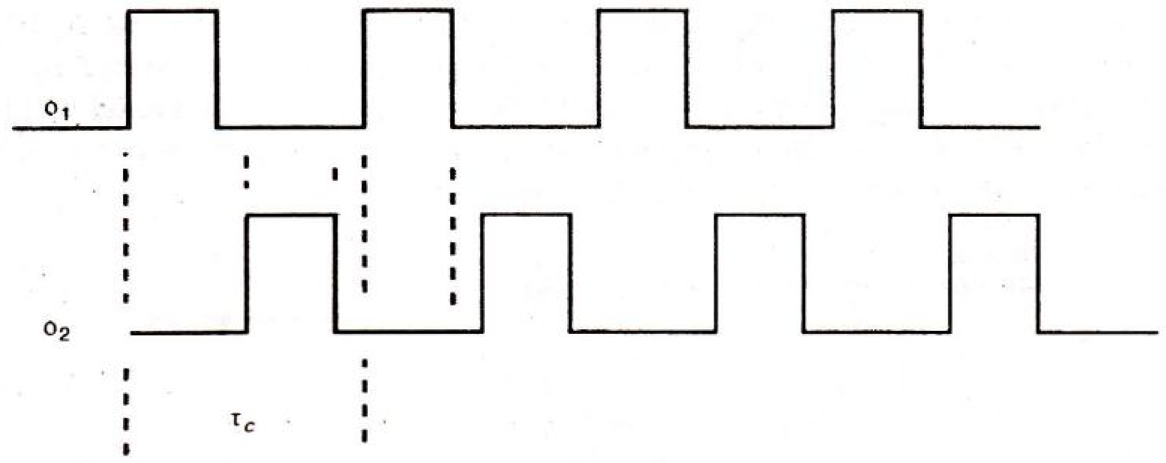
INPUT PROGRAMMING				FUNCTION $Z(A,B)$	
C_3	C_2	C_1	C_0		
0	0	0	0	0	$Z = 0$
0	0	0	1	$\bar{A}.\bar{B}$; $\overline{A+B}$	*Nor*
0	0	1	0	$A.\bar{B}$	
0	0	1	1	$\bar{B}$	Not B
0	1	0	0	$\bar{A}.B$	
0	1	0	1	$\bar{A}$	Not A
0	1	1	0	$A.\bar{B}+\bar{A}.B$	*Exclusive-Or*
0	1	1	1	$\bar{A}+\bar{B}$; $\overline{AB}$	*Nand*
1	0	0	0	$A.B$	*And*
1	0	0	1	$\bar{A}.\bar{B}+A.B$	Comparator
1	0	1	0	A	$O/P = A$
1	0	1	1	$A+\bar{B}$	
1	1	0	0	B	$O/P = B$
1	1	0	1	$\bar{A}+B$	
1	1	1	0	$A + B$	*Or*
1	1	1	1	1	$Z = 1$

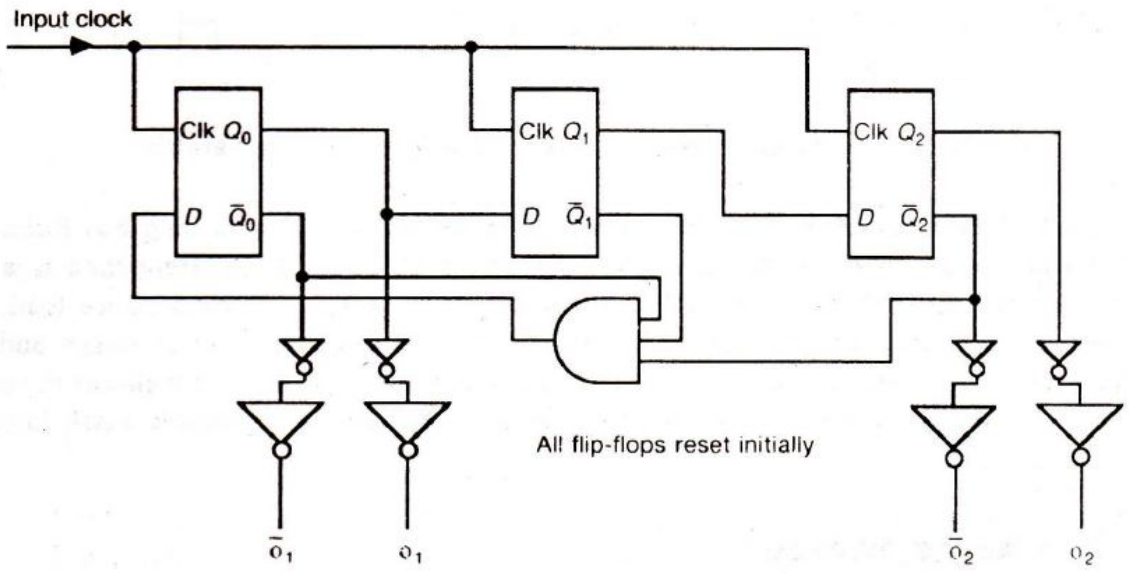
General logic function block (two variables).

CLOCKED SEQUENTIAL CIRCUITS:

Two -phase Clocking:

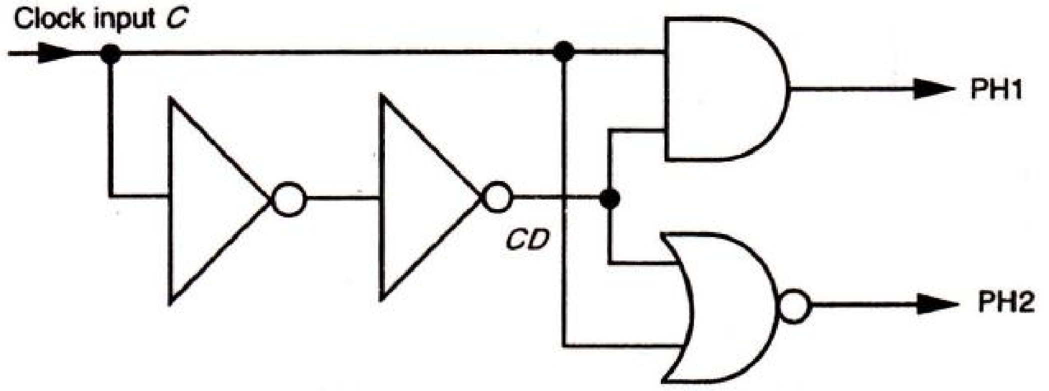
The clocked circuits to be considered here will be based on a two-phase non-overlapping clock signal.A two-phase clock offers a great deal of freedom in sequential circuit design if theclock period and the duration of the signals ϕ_1 and ϕ_2 are correctly chosen. If this is the case,data is allowed to become stable before any further transfer takes place and there is no chance of race conditions occurring.Clocked circuitry is considerably easier to design than the corresponding asynchronous sequential circuitry. It does, however, usually pay the penalty of being slower. However, at this stage of learning VLSI design we will concentrate on two-phase clocked sequential circuits alone and thus simplify design procedures.



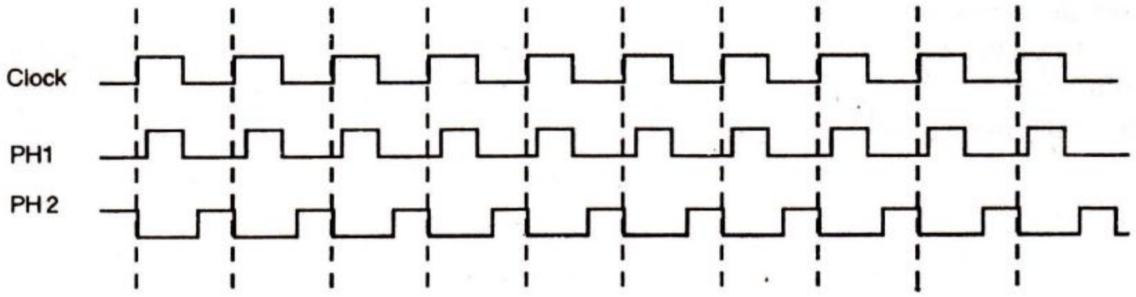


Two-phase clock generator using D flip-flops.

A very simple arrangement using combinational logic and generating a two-phase clock at the frequency of a single-phase input clock is set out in Figure. The input clock signal *C* is used to provide a delayed version of itself (CD) by passing it through an even number of inverters. The delay thus produced determines the underlap period for the two phase clock. Waveforms are as shown in Figure



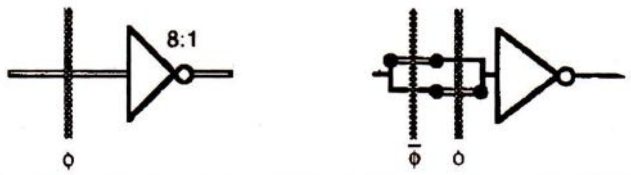
Simple two-phase clock generator circuit—basic form.



Waveforms for two-phase clock generator.

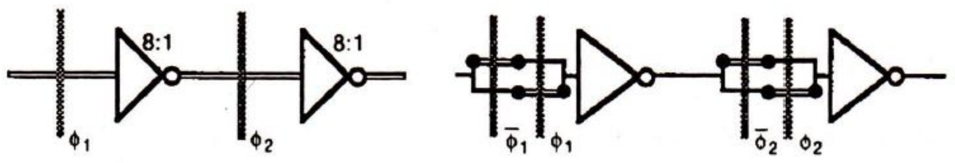
Dynamic Register Element:

The basic dynamic register element is shown in Figure in mixed stick/circuit notation and may be seen to consist of three transistors for nMOS and four for CMOS per stored bit in complemented form. The element's operation is simple to appreciate. $(V_{in})_t$ is clocked in by ϕ_1 (or ϕ_2) of the clock and charges the gate capacitance C_g of the inverter to V_{in}. If subscript t is taken to represent the time during which ϕ_1 (say) is at logic 1 and subscript $t + 1$ is taken to indicate the period during which ϕ_1 is at logic 0, then the available output will be $(\overline{V}_{in})_{t+1}$ which will be maintained by the stored charge on the gate until C_g discharges or until the next ϕ_1 signal occurs.



(a) nMOS pass transistor switched (b) CMOS transmission gate switched

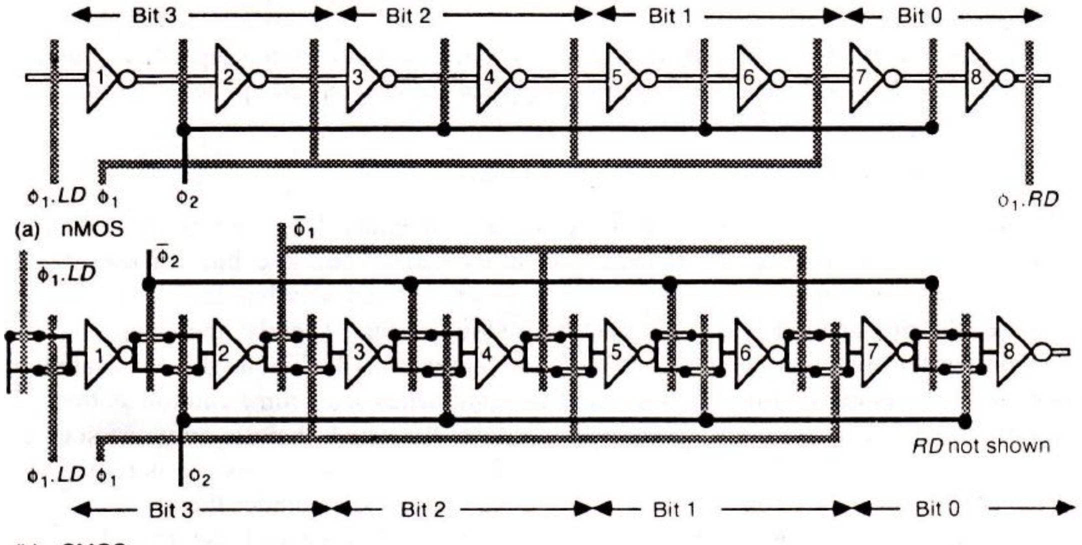
If uncomplemented storage is essential, the basic element is modified as indicated in Figure and will be seen to consist of six transistors for nMOS and eight for CMOS. Data clocked in on ϕ_1 is stored on C_{g1} and the corresponding output appears at the output of inverter 1. On ϕ_2 this value is clocked into and stored by C_{g2} and the output of inverter 2 then presents the 'true' form of the stored bit. Note that data read in on ϕ_1 is not available at the output until sometime following the next positive edge of the clock signal ϕ_2.



(a) nMOS pass transistor switched (b) CMOS transmission gate switched

DYNAMIC SHIFT REGISTER:

Cascading the basic elements of Figure gives a serial shift register arrangement which may be extended to n bits. A four-bit serial right shift nMOS register is illustrated in Figure Data bits are shifted in when $\phi_1.LD$ is present, one bit being entered on each ϕ_1 signal (provided that LD is logic 1). Each bit is stored in C_{g1} as it is entered, and then transferred complemented into C_{g2} during the next ϕ_2. Thus, after a ϕ_1 followed by ϕ_2 signal, the stored bit is present at the output of inverter 2. On the next ϕ_1, the next input bit is stored in C_{g1} and simultaneously the first bit stored is passed on to inverter pair 3 and 4 by being stored in C_{g3}, and so on. It will be seen that bits are thus clocked to the right along the shift register on each ϕ_1 followed by ϕ_2 sequence. Once four bits are stored, the data is available in parallel form at the outputs of inverters 2, 4, 6 and 8, and is also available in serial form from the output of inverter 8 when $\phi_1.RD$ is high as further clock sequences are received (where RD is the serial read control signal). The operation of the CMOS version is similar, transmission gates replacing inter-stage pass transistors and C_{in1} replacing C_{g1}, etc., as the storage capacitance.



Four-bit dynamic shift registers (nMOS and CMOS).

UNIT –IV

Programmable Logic Devices (PLDs)

Introduction:

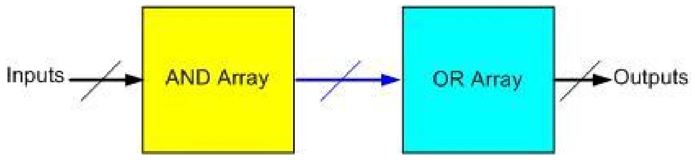
An IC that contains large numbers of gates, flip-flops, etc. that can be *configured by the user* to perform different functions is called a ***Programmable Logic Device (PLD)***.

The internal logic gates and/or connections of PLDs can be changed/configured by a programming process.

One of the simplest programming technologies is to use fuses. In the original state of the device, all the fuses are intact.

Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function.

PLDs are typically built with an *array* of AND gates (AND-array) and an *array* of OR gates (OR-array).



Advantages of PLDs:

Problems of using standard ICs:

Problems of using standard ICs in logic design are that they require hundreds or thousands of these ICs, considerable amount of circuit board space, a great deal of time and cost in inserting, soldering, and testing. Also require keeping a significant inventory of ICs.

Advantages of using PLDs:

Advantages of using PLDs are less board space, faster, lower power requirements (i.e., smaller power supplies), less costly assembly processes, higher reliability (fewer ICs and circuit connections means easier troubleshooting), and availability of design software.

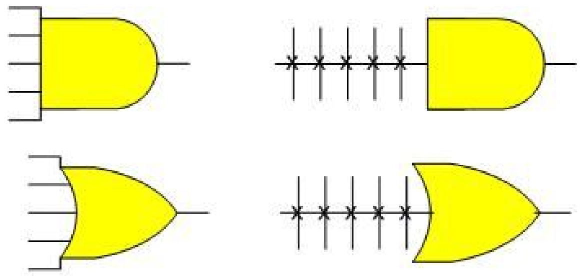
There are three fundamental types of standard PLDs: ***PROM, PAL,*** and ***PLA***.

A fourth type of PLD, which is discussed later, is the ***Complex Programmable Logic Device (CPLD),*** e.g., ***Field Programmable Gate Array (FPGA).***

A typical PLD may have hundreds to millions of gates.

In order to show the internal logic diagram for such technologies in a concise form, it is necessary to have special symbols for array logic.

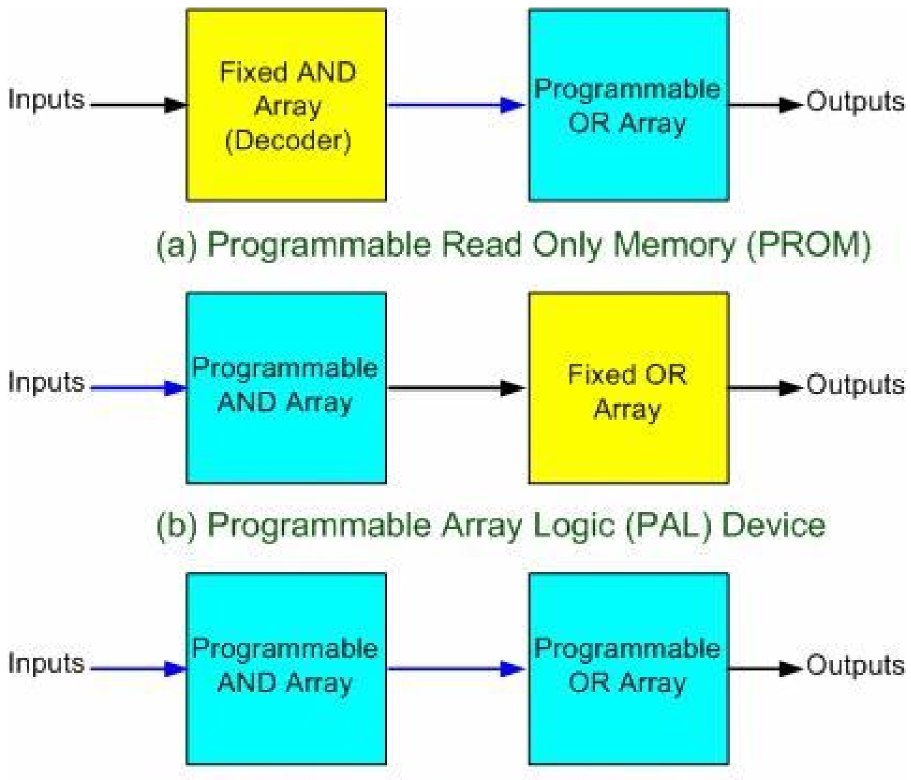
Figure shows the conventional and array logic symbols for a multiple input AND and a multiple input OR gate.



(a) Conventional Symbol (b) Array Logic Symbol

Three Fundamental Types of PLDs:

The three fundamental types of PLDs differ in the placement of programmable connections in the AND-OR arrays. Figure shows the locations of the programmable connections for the three types.



(c) Programmable Logic Array (PLA) Device

The ***PROM (Programmable Read Only Memory)*** has a fixed AND array (constructed as a decoder) and programmable connections for the output OR gates array. The PROM implements Boolean functions in sum-of-minterms form.

The ***PAL (Programmable Array Logic)*** device has a programmable AND array and fixed connections for the OR array.

The ***PLA (Programmable Logic Array)*** has programmable connections for both AND and OR arrays. So it is the most flexible type of PLD.

The ROM (Read Only Memory) or PROM (Programmable Read Only Memory):

The input lines to the AND array are hard-wired and the output lines to the OR array are programmable.

Each AND gate generates one of the possible AND products (i.e., minterms).

In the previous lesson, you have learnt how to implement a digital circuit using ROM.

The PLA (Programmable Logic Array):

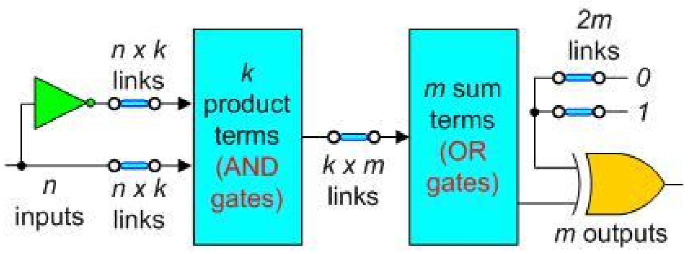
In PLAs, instead of using a decoder as in PROMs, a number (***k***) of AND gates is used where $k < 2^n$, (***n*** is the number of inputs).

Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM.

The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them.

The specific Boolean functions are implemented in sum of products form by opening appropriate links and leaving the desired connections.

A block diagram of the PLA is shown in the figure. It consists of ***n*** inputs, ***m*** outputs, and ***k*** product terms.



The product terms constitute a group of ***k*** AND gates each of *2n* inputs.

Links are inserted between all ***n*** inputs and their complement values to each of the AND gates.

Links are also provided between the outputs of the AND gates and the inputs of the OR gates.

Since PLA has ***m***-outputs, the number of OR gates is ***m***.

The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the **true** form or in the **complement** form.

The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = X^{/}$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$).

Thus, the total number of programmable links is ***2n* x *k* + *k* x *m* + *2m***.

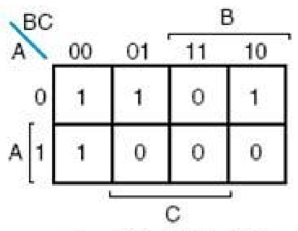
The size of the PLA is specified by the number of inputs ***(n)***, the number of product terms ***(k)***, and the number of outputs ***(m)***, (the number of sum terms is equal to the number of outputs).

Example:

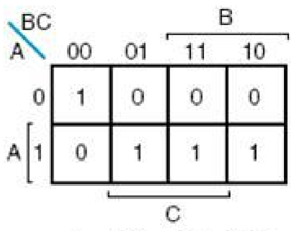
Implement the combinational circuit having the shown truth table, using PLA.

A	B	C	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.



$$F_1 = \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{B}\overline{C}$$
$$\overline{F_1} = AB + AC + BC$$



$$F_2 = AB + AC + \overline{A}\overline{B}\overline{C}$$
$$\overline{F_2} = \overline{A}C + \overline{A}B + A\overline{B}\overline{C}$$

Designing using a PLA, a careful investigation must be taken in order to reduce the distinct product terms. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

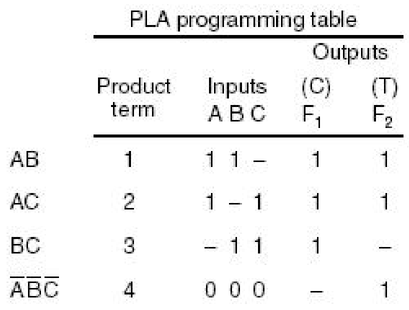
The combination that gives a minimum number of product terms is:

$F_1' = AB + AC + BC$ *or* $F_1 = (AB + AC + BC)'$

$F_2 = AB + AC + A'B'C'$

This gives only 4 distinct product terms: *AB, AC, BC,* and *A'B'C'*.

So the PLA table will be as follows:



PLA programming table

	Product term	Inputs A B C	Outputs (C) F_1	Outputs (T) F_2
AB	1	1 1 –	1	1
AC	2	1 – 1	1	1
BC	3	– 1 1	1	–
$\bar{A}\bar{B}\bar{C}$	4	0 0 0	–	1

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1.

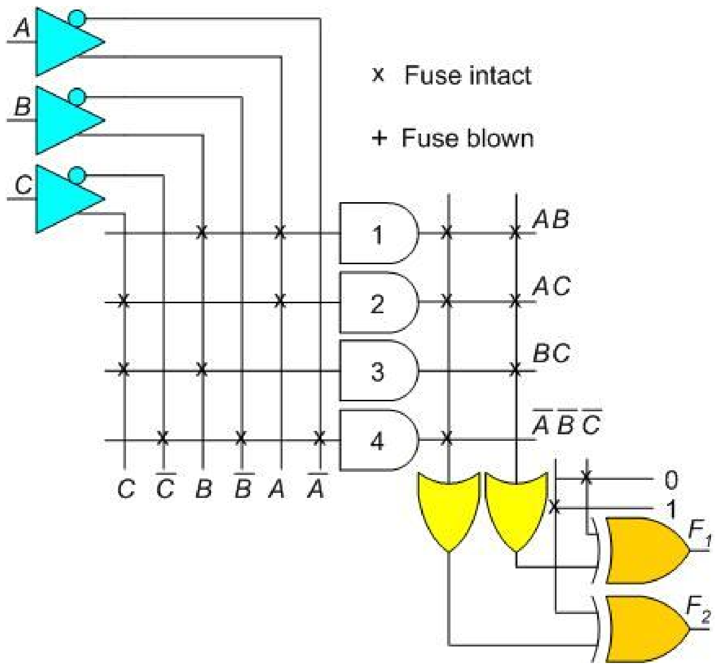
A 1 in the **Inputs** column specifies a path from the corresponding input to the input of the AND gate that forms the product term.

A 0 in the **Inputs** column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.

The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input.

In the Outputs column, a **T (true)** specifies that the other input of the corresponding XOR gate can be connected to 0, and a **C (complement)** specifies a connection to 1.

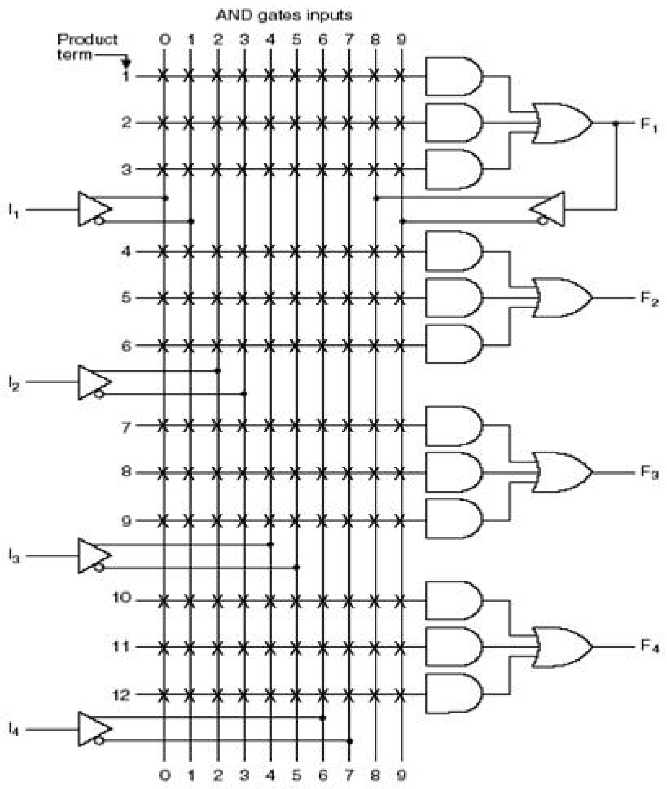
Note that output ***F_1*** is the normal (or true) output even though a **C** (for complement) is marked over it. This is because ***F_1'*** is generated with AND-OR circuit prior to the output XOR. The output XOR complements the function *F_1'* to produce the true *F_1* output as its second input is connected to logic 1.



The PAL (Programmable Array Logic):

The PAL device is a PLD with a fixed OR array and a programmable AND array.

As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA.



The device shown in the figure has 4 inputs and 4 outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate.

The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section.

Each AND gate has 10 programmable input connections indicating by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate.

One of the outputs ***F_1*** is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections.

Designing using a PAL device, the Boolean functions must be simplified to fit into each section.

The number of product terms in each section is fixed and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function.

Example:

Implement the following Boolean functions using the PAL device as shown above:

$W(A, B, C, D) = \sum m(2, 12, 13)$

$X(A, B, C, D) = \sum m(7, 8, 9, 10, 11, 12, 13, 14, 15)$

$Y(A, B, C, D) = \sum m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$

$Z(A, B, C, D) = \sum m(1, 2, 8, 12, 13)$

Simplifying the 4 functions to a minimum number of terms results in the following

Boolean functions:

$W = ABC' + A'B'CD'$

$X = A + BCD$

$Y = A'B + CD + B'D'$

$Z = ABC' + A'B'CD + AC'D' + A'B'C'D$

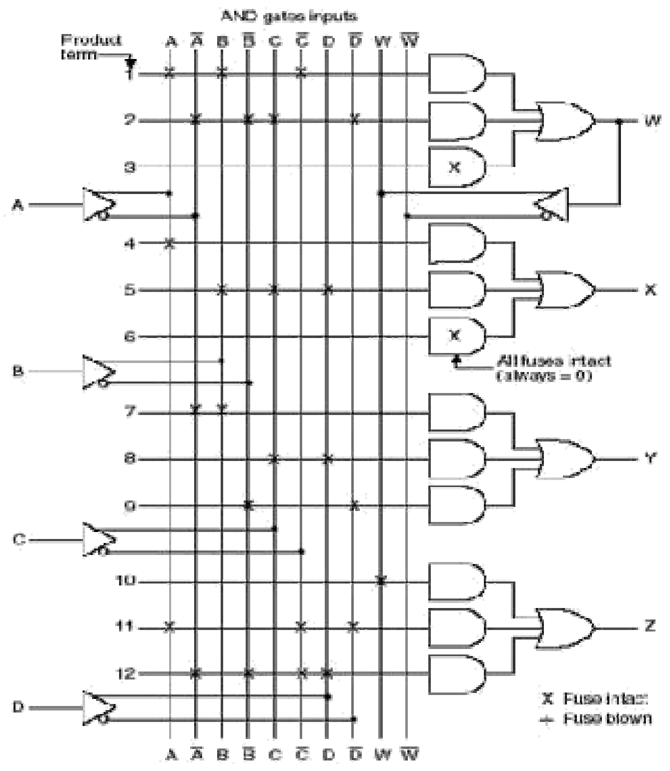
$\quad = W + AC'D' + A'B'C'D$

Note that the function for ***Z*** has four product terms. The logical sum of two of these terms is equal to ***W***. Thus, by using ***W***, it is possible to reduce the number of terms for ***Z*** from four to three, so that the function can fit into the given PAL device.

The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed.

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	—	—	$W = AB\overline{C}$
2	0	0	1	0	—	$+\overline{A}\,\overline{B}C\overline{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$
8	—	—	1	1	—	$+CD$
9	—	0	—	0	—	$+\overline{B}\,\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+A\overline{C}\,\overline{D}$
12	0	0	0	1	—	$+\overline{A}\,\overline{B}\,\overline{C}D$

The figure shows the connection map for the PAL device, as specified in the programming table.

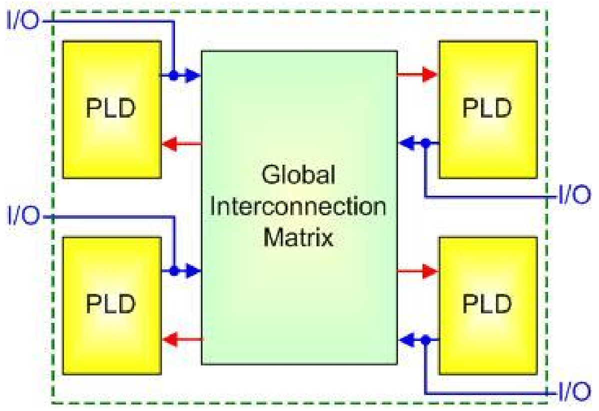


Since both W and X have two product terms, third AND gate is not used. If all the inputs to this AND gate left intact, then its output will always be 0, because it receives both the true and complement of each input variable i.e., AA' =0

Complex Programmable Logic Devices (CPLDs):

A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix.

Thus a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.



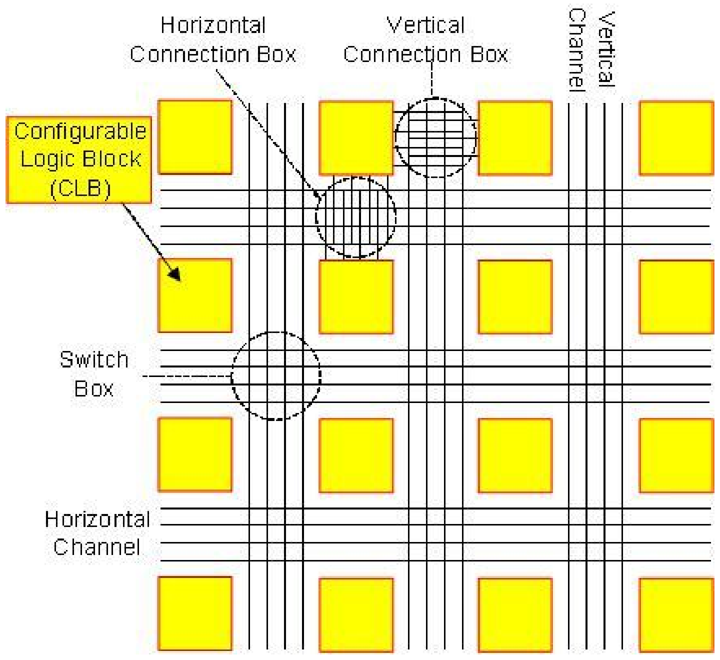
Field Programmable Gate Arrays (FPGAs):

The FPGA consists of 3 main structures:

1. Programmable logic structure,
2. Programmable routing structure, and
3. Programmable Input/Output (I/O).

1. Programmable logic structure

The programmable logic structure FPGA consists of a 2-dimensional array of configurable logic blocks (CLBs).



Each CLB can be configured (programmed) to implement *any* Boolean function of its input variables. Typically CLBs have between 4-6 input variables. Functions of larger number of variables are implemented using more than one CLB.

In addition, each CLB typically contains 1 or 2 FFs to allow implementation of sequential logic.

Large designs are partitioned and mapped to a number of CLBs with each CLB configured (programmed) to perform a particular function.

These CLBs are then connected together to fully implement the target design.

Connecting the CLBs is done using the FPGA programmable routing structure.

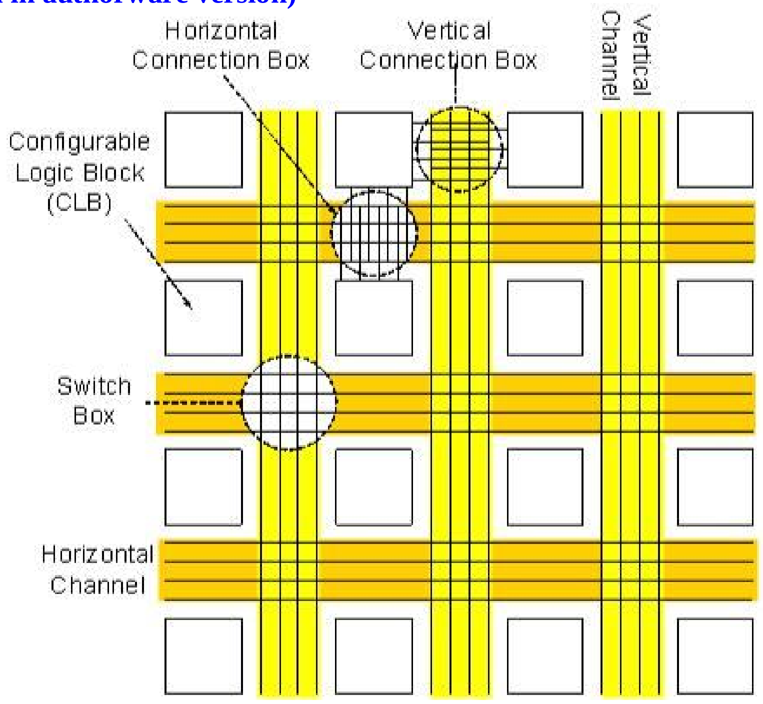
2. Programmable routing structure

To allow for flexible interconnection of CLBs, FPGAs have 3 *programmable* routing resources:

1. Vertical and horizontal routing channels which consist of different length wires that can be connected together if needed. These channel run vertically and horizontally between columns and rows of CLBs as shown in the Figure.

2. Connection boxes, which are a set of programmable links that can connect input and output pins of the CLBs to wires of the vertical or the horizontal routing channels.

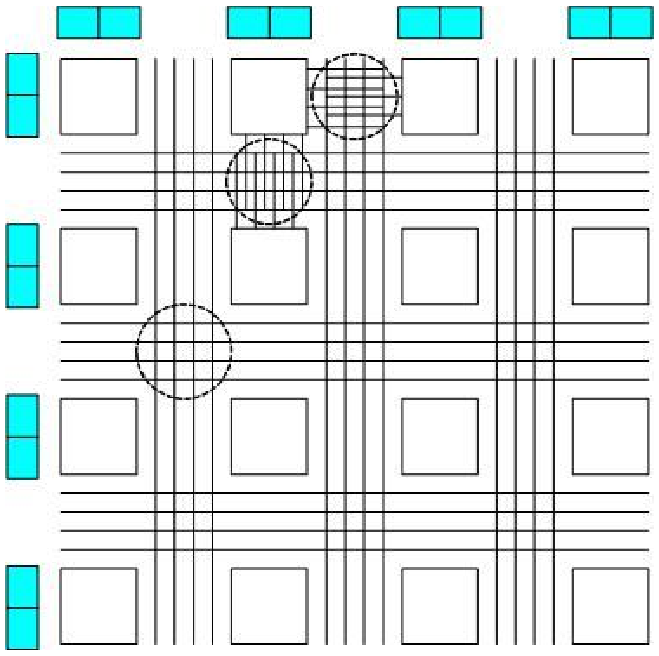
3. Switch boxes, located at the intersection of the vertical and horizontal channels. These are a set of programmable links that can connect wire segments in the horizontal and vertical channels. **(see animation in authorware version)**



3. Programmable I/O

These are mainly buffers that can be configured either as input buffers, output buffers or input/output buffers.

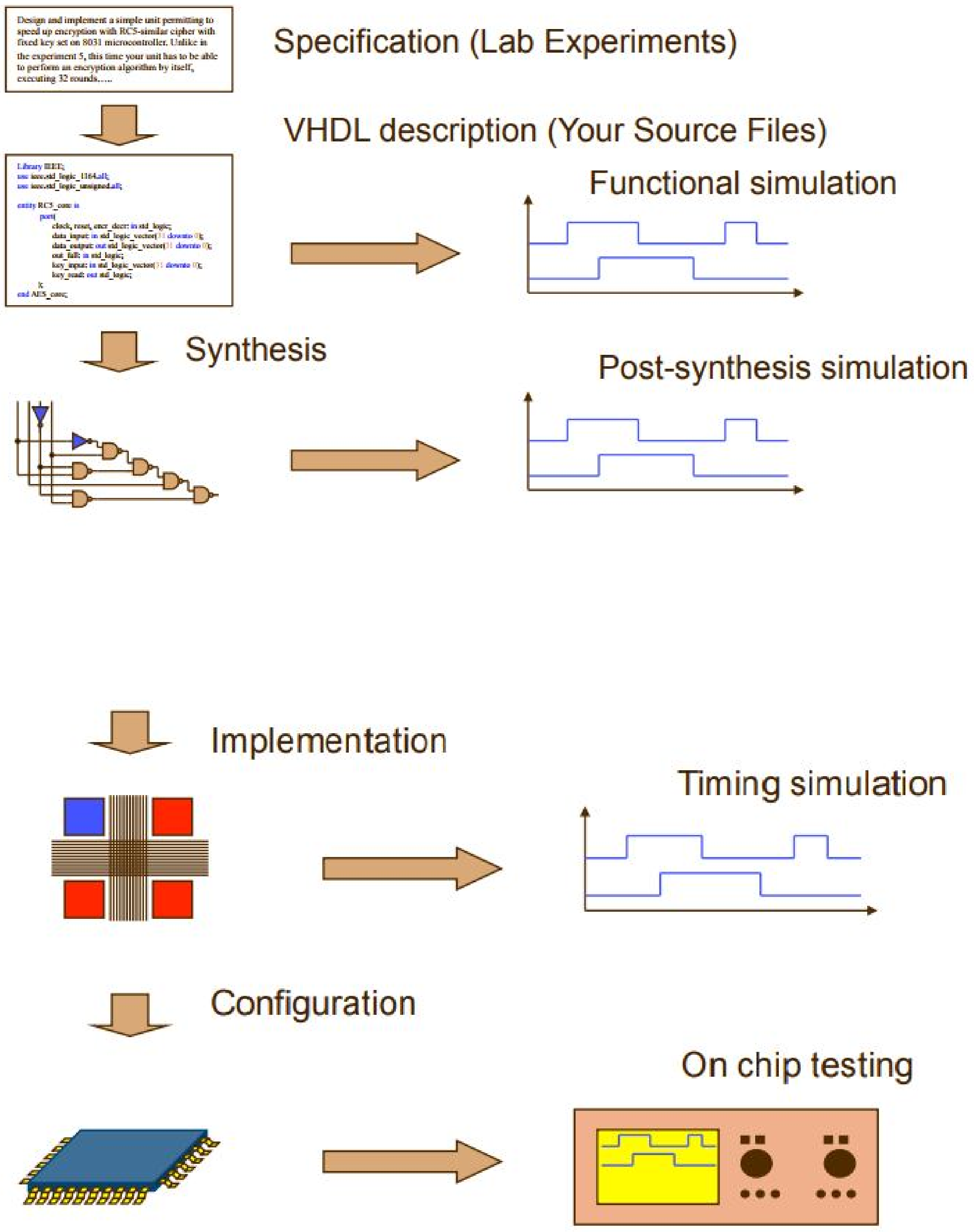
They allow the pins of the FPGA chip to function either as input pins, output pins or input/output pins.



Programmable

I/Os

FPGA DESIGN FLOW


Design and implement a simple unit permitting to speed up encryption with RC5-similar cipher with fixed key set on 8031 microcontroller. Unlike in the experiment 5, this time your unit has to be able to perform an encryption algorithm by itself, executing 32 rounds…..

```
Library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity RC5_core is
        port(
            clock, reset, encr_decr: in std_logic;
            data_input: in std_logic_vector(31 downto 0);
            data_output: out std_logic_vector(31 downto 0);
            out_full: in std_logic;
            key_input: in std_logic_vector(31 downto 0);
            key_read: out std_logic;
        );
end AES_core;
```

Specification (Lab Experiments)
VHDL description (Your Source Files)
Functional simulation
Synthesis
Post-synthesis simulation
Implementation
Timing simulation
Configuration
On chip testing

Design Entry:

There are different techniques for design entry. Schematic based, Hardware Description Language and combination of both etc. . Selection of a method depends on the design and designer. If the designer wants to deal more with Hardware, then Schematic entry is the better choice. When the design is complex or the designer thinks the design in an algorithmic way then HDL is the better choice. Language based entry is faster but lag in performance and density. HDLs represent a level of abstraction that can isolate the designers from the details of the hardware implementation. Schematic based entry gives designers much more visibility into the hardware. It is the better choice for those who are hardware oriented.

Synthesis :

The process which translates VHDL or Verilog code into a device netlist formate. i.e a complete circuit with logical elements(gates, flip flops, etc…) for the design.If the design contains more than one sub designs, ex. to implement a processor, we need a CPU as one design element and RAM as another and so on, then the synthesis process generates netlist for each design element Synthesis process will check code syntax and analyze the hierarchy of the design which ensures that the design is optimized for the design architecture, the designer has selected**.**

Implementation:

This process consists a sequence of three steps 1. Translate 2. Map 3. Place and Route Translate process combines all the input netlists and constraints to a logic design file. This information is saved as a NGD (Native Generic Database) file. This can be done using NGD Build program. Here, defining constraints is nothing but, assigning the ports in the design to the physical elements (ex. pins, switches, buttons etc) of the targeted device and specifying time requirements of the design. This information is stored in a file named UCF (User Constraints File). Tools used to create or modify the UCF are PACE, Constraint Editor etc

Map:

This process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an NCD (Native Circuit Description) file which physically represents the design mapped to the components of FPGA. MAP program is used for this purpose.

Place and Route:

PAR program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. Ex. if a sub block is placed in a logic block which is very near to IO pin, then it may save the time but it may effect some other constraint. So trade off between all the constraints is taken account by the place and route process The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consists the routing information.

Structure of a basic FPGA cell (Configurable Logic Block)

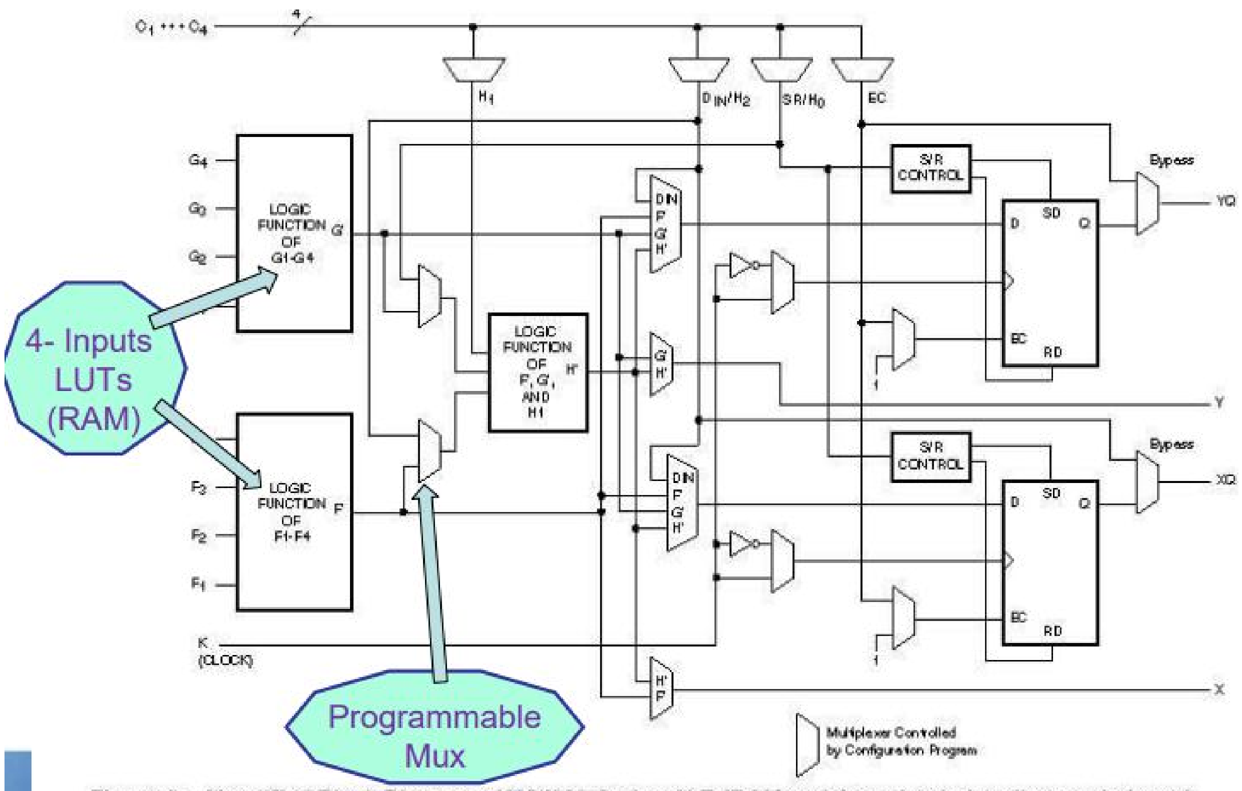
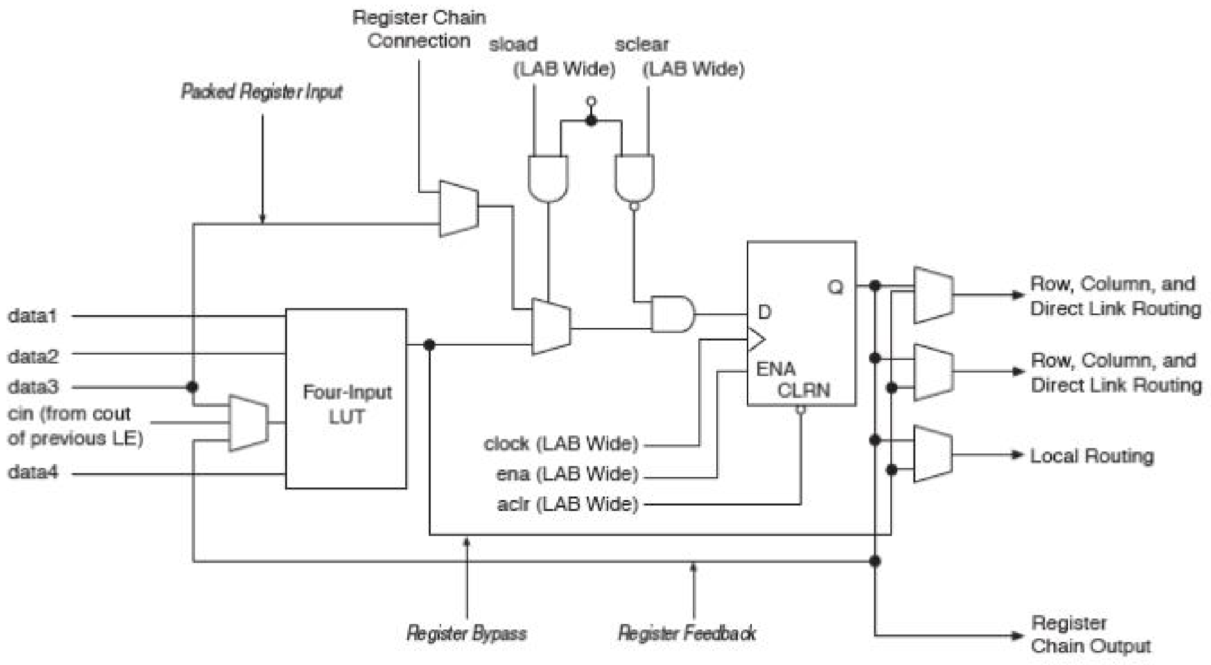
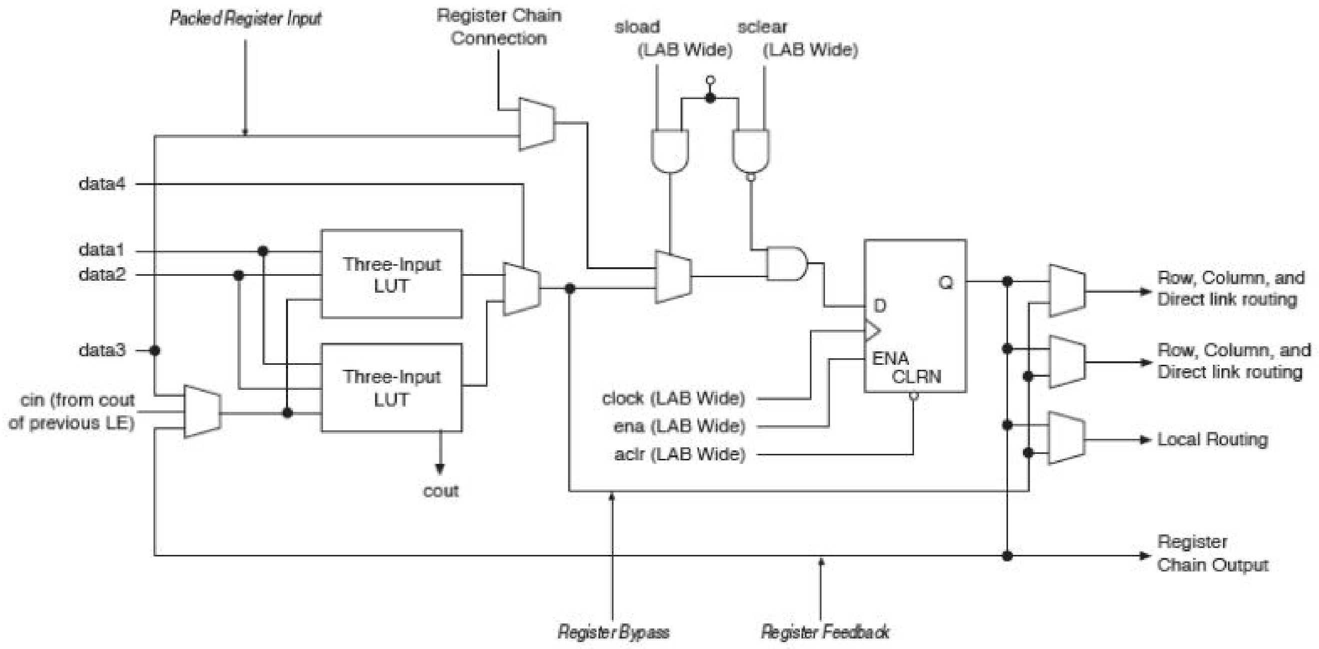


Figure 1: Simplified Block Diagram of XC4000-Series CLB (RAM and Carry Logic functions not shown)

Altera Cyclone III
Logic Element (LE) – Normal Mode



Altera Cyclone III
Logic Element (LE) – Arithmetic Mode



UNIT V

HARDWARE DESCRIPTION LANGUAGES

Introduction

With the advent of VLSI technology and increased usage of digital circuits, designers has to design single chips with millions of transistors. It became almost impossible to verify these circuits of high complexity on breadboard. Hence Computer-aided techniques became critical for verification and design of VLSI digital circuits.As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further. All these factors which led to the evolution of Computer-Aided Digital Design, intern led to the emergence of Hardware Description Languages.
Verilog HDL and VHDL are the popular HDLs.Today, Verilog HDL is an accepted IEEE standard. In 1995, the original standard IEEE 1364-1995 was approved. IEEE 1364-2001 is the latest Verilog HDL standard that made significant improvements to the original standard.
Specifications comes first, they describe abstractly the functionality, interface, and the architecture of the digital IC circuit to be designed.

- Behavioral description is then created to analyze the design in terms of functionality, performance, compliance to given standards, and other specifications.
- RTL description is done using HDLs. This RTL description is simulated to test functionality. From here onwards we need the help of EDA tools.
- RTL description is then converted to a gate-level net list using logic synthesis tools. A gate-level netlist is a description of the circuit in terms of gates and connections between them, which are made in such a way that they meet the timing, power and area specifications.
- Finally a physical layout is made, which will be verified and then sent to fabrication.

Importance of HDLs
RTL descriptions, independent of specific fabrication technology can be made an verified.functional verification of the design can be done early in the design cycle.

- Better representation of design due to simplicity of HDLs when compared to gate-level schematics.
- Modification and optimization of the design became easy with HDLs.
- Cuts down design cycle time significantly because the chance of a functional bug at a later stage in the design-flow is minimal.

Verilog HDL

Verilog HDL is one of the most used HDLs. It can be used to describe designs at four levels of abstraction:

1. Algorithmic level.
2. Register transfer level (RTL).

3. Gate level.
4. Switch level (the switches are MOS transistors inside gates).

Why Verilog ?

- Easy to learn and easy to use, due to its similarity in syntax to that of the C programming language.
- Different levels of abstraction can be mixed in the same design.
- Availability of Verilog HDL libraries for post-logic synthesis simulation.
- Most of the synthesis tools support Verilog HDL.
- The *Programming Language Interface (PLI)* is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

Digital design methods

Digital design methods are of two types:

1. ***Top-down* design method** : In this design method we first define the top-level block and then we build necessary sub-blocks, which are required to build the top-level block. Then the sub-blocks are divided further into smaller-blocks, and so on. The bottom level blocks are called as leaf cells. By saying bottom level it means that the leaf cell cannot be divided further.
2. ***Bottom-up* design method** : In this design method we first find the bottom leaf cells, and then start building upper sub-blocks and building so on, we reach the top-level block of the design.

In general a combination of both types is used. These types of design methods helps the design architects, logics designers, and circuit designers. Design architects gives specifications to the logic designers, who follow one of the design methods or both. They identify the leaf cells. Circuit designers design those leaf cells, and they try to optimize leaf cells in terms of power, area, and speed. Hence all the design goes parallel and helps finishing the job faster.

Operators

There are three types of operators: unary, binary, and ternary, which have one, two, and three operands respectively.

Unary : Single operand, which precede the operand.
Ex: x = ~y
~ is a unary operator
y is the operand

binary : Comes between two operands.
Ex: x = y || z
|| is a binary operator
y and z are the operands

ternary : Ternary operators have two separate operators that separate three operands.

Ex: p = x ? y : z

? : is a ternary operator

x, y, and z are the operands

List of operators is given here.

Comments

Verilog HDL also have two types of commenting, similar to that of C programming language. // is used for single line commenting and '/*' and '*/' are used for commenting multiple lines which start with /* and end with */.

EX: // single line comment

/* Multiple line

commenting */

/* This is a // LEGAL comment */

/* This is an /* ILLEGAL */ comment */

Whitespace

- - \b - backspace
- - \t - tab space
- - \n - new line

In verilog Whitespace is ignored except when it separates tokens. Whitespace is not ignored in strings. Whitesapces are generally used in writing test benches.

Strings

A string in verilog is same as that of C programming language. It is a sequence of characters enclosed in double quotes. String are treated as sequence of one byte ASCII values, hence they can be of one line only, they cannot be of multiple lines.

Ex: " This is a string "

" This is not treated as

string in verilog HDL "

Identifiers

Identifiers are user-defined words for variables, function names, module names, block names and instance names.Identifiers begin with a letter or underscore and can include any number of letters, digits and underscores. It is not legal to start identifiers with number or the dollar($) symbol in Verilog HDL. Identifiers in Verilog are case-sensitive.

Keywords

Keywords are special words reserved to define the language constructs. In verilog all keywords are in lowercase only. A list of all keywords in Verilog is given below:

always	event	output	strong1
and	for	parameter	supply0
assign	force	pmos	supply1
attribute	forever	posedge	table
begin	fork	primitive	task
buf	function	pull0	time
bufif0	highz0	pull1	tran
bufif1	highz1	pulldown	tranif0
case	if	pullup	tranif1
casex	ifnone	rcmos	tri
casez	initial	real	tri0
cmos	inout	realtime	tri1
deassign	input	reg	triand
default	integer	release	trior
defparam	join	repeat	trireg
disable	medium	rnmos	unsigned
edge	module	rpmos	vectored
else	large	rtran	wait
end	macromodule	rtranif0	wand
endattribute	nand	rtranif1	weak0
endcase	negedge	scalared	weak1
endfunction	nmos	signed	while
endmodule	nor	small	wire
endprimitive	not	specify	wor
endspecify	notif0	specparam	xnor
endtable	notif1	strength	xor
endtask	or	strong0	

Verilog keywords also includes compiler directives, system tasks, and functions. Most of the keywords will be explained in the later sections.

Number Specification

Sized Number Specification

Representation: *[size]'[base][number]*

- [size] is written only in decimal and specifies the number of bits.

- [base] could be 'd' or 'D' for decimal, 'h' or 'H' for hexadecimal, 'b' or 'B' for binary, and 'o' or 'O' for octal.
- [number] The number is specified as consecutive digits. Uppercase letters are legal for number specification (in case of hexadecimal numbers).

Ex: 4'b1111 : 4-bit binary number
16'h1A2F : 16-bit hexadecimal number
32'd1 : 32-bit decimal number
8'o3 : 8-bit octal number

Unsized Number Specification

By default numbers that are specified without a [base] specification are decimal numbers. Numbers that are written without a [size] specification have a default number of bits that is simulator and/or machine specific (generally 32).

Ex: 123 : This is a decimal number
'hc3 : This is a hexadecimal number
Number of bits depends on simulator/machine, generally 32.

x or z values

x **-** Unknown value.
z **-** High impedance value
An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base.

Note: If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative Numbers

Representation: -*[size]'[base][number]*

Ex: -8'd9 : 8-bit negative number stored as 2's complement of 8
-8'sd3 : Used for performing signed integer math
4'd-2 : Illegal

Underscore(_) and question(?) mark

An underscore, "**_**" is allowed to use anywhere in a number except in the beginning. It is used only to improve readability of numbers and are ignored by Verilog. A question mark "**?**" is the alternative for z w.r.t. numbers

Ex: 8'b1100_1101 : Underscore improves readability
4'b1??1 : same as 4'b1zz1

Value Set

The Verilog HDL value set consists of four basic values:

- 0 **-** represents a logic zero, or a false condition.
- 1 **-** represents a logic one, or a true condition.
- x **-** represents an unknown logic value.
- z **-** represents a high-impedance state.

The values 0 and 1 are logical complements of one another. Almost all of the data types in the Verilog HDL store all four basic values.

Nets

Nets are used to make connections between hardware elements. Nets simply reflect the value at one end(head) to the other end(tail). It means the value they carry is continuously driven by the output of a hardware element to which they are connected to. Nets are generally declared using the keyword *wire*. The default value of net (wire) is z. If a net has no driver, then its value is **z**.

Registers

Registers are data storage elements. They hold the value until they are replaced by some other value. Register doesn't need a driver, they can be changed at anytime in a simulation. Registers are generally declared with the keyword *reg*. Its default value is x. Register data types should not be confused with hardware registers, these are simply variables.

Integers

Integer is a register data type of 32 bits. The only difference of declaring it as integer is that, it becomes a signed value. When you declare it as a 32 bit register (array) it is an unsigned value. It is declared using the keyword *integer*.

Real Numbers

Real number can be declared using the keyword *real*. They can be assigned values as follows:
real r_1;

r_1 = 1.234; // Decimal notation.
r_1 = 3e4; // Scientific notation.

Parameters

Parameters are the constants that can be declared using the keyword *parameter*. Parameters are in general used for customization of a design. Parameters are declared as follows:

```
parameter p_1 = 123; // p_1 is a constant with value 123.
```

Keyword *defparam* can be used to change a parameter value at module instantiation. Keyword *localparam* is usedd to declare local parameters, this is used when their value should not be changed.

Vectors

Vectors can be a net or reg data types. They are declared as [high:low] or [low:high], but the left number is always the MSB of the vector.

```
wire [7:0] v_1; // v_1[7] is the MSB.
reg [0:15] v_2; // v_2[15] is the MSB.
```

In the above examples: If it is written as v_1[5:2], it is the part of the entire vector which contains 4 bits in order: v_1[5], v_1[4], v_1[3], v_1[2]. Similarly v_2[0:7], means the first half part of the vecotr v_2.
Vector parts can also be specified in a different way:
vector_name[start_bit+:width] : part-select increments from start_bit. In above example: v_2[0:7] is same as v_2[0+:8].
vector_name[start_bit-:width] : part-select decrements from start_bit. In above example: v_1[5:2] is same as v_1[5-:4].

Arrays

Arrays of reg, integer, real, time, and vectors are allowed. Arrays are declared as follows:

```
reg a_1[0:7];
real a_3[15:0];
wire [0:3] a_4[7:0]; // Array of vector
integer a_5[0:3][6:0]; // Double dimensional array
```

Strings

Strings are register data types. For storing a character, we need a 8-bit register data type. So if you want to create string variable of length *n*. The string should be declared as register data type of length *n*8*.

```
reg [8*8-1:0] string_1; // string_1 is a string of length 8.
```

Time Data Type

Time data type is declared using the keyword *time*. These are generally used to store simulation time. In general it is 64-bit long.

```
time t_1;
t_1 = $time; // assigns current simulation time to t_1.
```

There are some other data types, but are considered to be advanced data types, hence they are not discussed here.

A module is the basic building block in Verilog HDL. In general many elements are grouped to form a module, to provide a common functionality, which can be used at many places in the design. Port interface (using input and output ports) helps in providing the necessary functionality to the higher-level blocks. Thus any design modifications at lower level can be easily implemented without affecting the entire design code. The structure of a module is show in the figure below.
Keyword ***module*** is used to begin a module and it ends with the keyword ***endmodule***. The syntax is as follows:

```
module module_name
---
// internals
---
endmodule
```

Example: D Flip-flop implementation (Try to understand the module structure, ignore unknown constraints/statements).

```
module D_FlipFlop(q, d, clk, reset);

// Port declarations
output q;
reg q;
input d, clk, reset;

// Internal statements - Logic
always @(posedge reset or poseedge clk)
if (reset)
q < = 1'b0;
else
q < = d;

// endmodule statement
endmodule
```

Note:

- Multiple modules can be defined in a single design file with any order.
- See that the endmodule statement should not written as endmodule; (no ; is used).
- All components except module, module name, and endmodule are optional.

- The 5 internal components can come in any order.

Modules communicate with external world using ports. They provide interface to the modules. A module definition contains list of ports. All ports in the list of ports must be declared in the module, ports can be one the following types:

- Input port, declared using keyword **input**.
- Output port, declared using keyword **output**.
- Bidirectional port, declared using keyword **inout**.

All the ports declared are considered to be as wire by default. If a port is intended to be a wire, it is sufficient to declare it as *output*, *input*, or *inout*. If output port holds its value it should be declared as *reg* type. Ports of type *input* and *inout* cannot be declared as *reg* because *reg* variables hold values and input ports should not hold values but simply reflect the changes in the external signals they are connected to.

Port Connection Rules

- Inputs: Always of type *net*(*wire*). Externally, they can be connected to *reg* or *net* type variable.
- Outputs: Can be of *reg* or *net* type. Externally, they must be connected to a *net* type variable.
- Bidirectional ports (*inout*): Always of type *net*. Externally, they must be connected to a *net* type variable.

Note:

- It is possible to connect internal and external ports of different size. In general you will receive a warning message for width mismatch.
- There can be unconnected ports in module instances.

Ports can declared in a module in C-language style:

```
module module_1( input a, input b, output c);
--
// Internals
--
endmodule
```

If there is an instance of above module, in some other module. Port connections can be made in two types.

Connection by Ordered List:

```
module_1 instance_name_1 ( A, B, C);
```

Connecting ports by name:

```
module_1 instance_name_2 (.a(A), .c(C), .b(B));
```

In connecting port by name, order is ignored.

Logical Operators

Symbol	Description	#Operators
!	Logical negation	One
||	Logical OR	Two
&&	Logical AND	Two

Relational Operators

Symbol	Description	#Operators
>	Greater than	Two
<	Less than	Two
>=	Greater than or equal to	Two
<=	Less than or equal to	Two

Equality Operators

Symbol	Description	#Operators
==	Equality	Two
!=	Inequality	Two
===	Case equality	Two
!==	Case inequality	Two

Arithmetic Operators

Symbol	Description	#Operators
+	Add	Two
-	Substract	Two
*	Multiply	Two
/	Divide	Two
**	Power	Two
%	Modulus	Two

Bitwise Operators

Symbol	Description	#Operators
~	Bitwise negation	One
&	Bitwise AND	Two
|	Bitwise OR	Two
^	Bitwise XOR	Two
^~ or ~^	Bitwise XNOR	Two

Reduction Operators

Symbol	Description	#Operators
&	Reduction AND	One
~&	Reduction NAND	One
|	Reduction OR	One
~|	Reduction NOR	One
^	Reduction XOR	One
^~ or ~^	Reduction XNOR	One

Shift Operators

Symbol	Description	#Operators
>>	Right shift	Two
<<	Left shift	Two
>>>	Arithmetic right shift	Two
<<<	Arithmetic left shift	Two

Conditional Operators

Symbol	Description	#Operators
?:	Conditional	Two

Replication Operators

Symbol	Description	#Operators
{ { } }	Replication	> One

Concatenation Operators

Symbol	Description	#Operators
{ }	Concatenation	> One

Operator Precedence

Introduction

In Verilog HDL a module can be defined using various levels of abstraction. There are four levels of abstraction in verilog. They are:

- Behavioral or algorithmic level: This is the highest level of abstraction. A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.
- Data flow level: In this level the module is designed by specifying the data flow. Designer must how data flows between various registers of the design.
- Gate level: The module is implemented in terms of logic gates and interconnections between these gates. Designer should know the gate-level diagram of the design.
- Switch level: This is the lowest level of abstraction. The design is implemented using switches/transistors. Designer requires the knowledge of switch-level implementation details.

Gate-level modeling is virtually the lowest-level of abstraction, because the switch-level abstraction is rarely used. In general, gate-level modeling is used for implementing lowest level modules in a design like, full-adder, multiplexers, etc. Verilog HDL has gate primitives for all basic gates.

Gate Primitives

Gate primitives are predefined in Verilog, which are ready to use. They are instantiated like modules. There are two classes of gate primitives: Multiple input gate primitives and Single input gate primitives.
Multiple input gate primitives include and, nand, or, nor, xor, and xnor. These can have multiple inputs and a single output. They are instantiated as follows:

```
// Two input AND gate.
and and_1 (out, in0, in1);

// Three input NAND gate.
nand nand_1 (out, in0, in1, in2);

// Two input OR gate.
```

or or_1 (out, in0, in1);

// Four input NOR gate.
nor nor_1 (out, in0, in1, in2, in3);

// Five input XOR gate.
xor xor_1 (out, in0, in1, in2, in3, in4);

// Two input XNOR gate.
xnor and_1 (out, in0, in1);

Note that instance name is not mandatory for gate primitive instantiation. The truth tables of multiple input gate primitives are as follows:

Single input gate primitives include not, buf, notif1, bufif1, notif0, and bufif0. These have a single input and one or more outputs. Gate primitives notif1, bufif1, notif0, and bufif0 have a control signal. The gates propagate if only control signal is asserted, else the output will be high impedance state (z). They are instantiated as follows:

// Inverting gate.
not not_1 (out, in);

// Two output buffer gate.
buf buf_1 (out0, out1, in);

// Single output Inverting gate with active-high control signal.
notif1 notif1_1 (out, in, ctrl);

// Double output buffer gate with active-high control signal.
bufif1 bufif1_1 (out0, out1, in, ctrl);

// Single output Inverting gate with active-low control signal.
notif0 notif0_1 (out, in, ctrl);

// Single output buffer gate with active-low control signal.
bufif0 bufif1_0 (out, in, ctrl);

The truth tables are as follows:

Array of Instances:

```
wire [3:0] out, in0, in1;
and and_array[3:0] (out, in0, in1);
```

The above statement is equivalent to following bunch of statements:

```
and and_array0 (out[0], in0[0], in1[0]);
and and_array1 (out[1], in0[1], in1[1]);
and and_array2 (out[2], in0[2], in1[2]);
and and_array3 (out[3], in0[3], in1[3]);
```

>> Examples

Gate Delays:

In Verilog, a designer can specify the gate delays in a gate primitive instance. This helps the designer to get a real time behavior of the logic circuit.

Rise delay: It is equal to the time taken by a gate output transition to 1, from another value 0, x, or z.

Fall delay: It is equal to the time taken by a gate output transition to 0, from another value 1, x, or z.

Turn-off delay: It is equal to the time taken by a gate output transition to high impedance state, from another value 1, x, or z.

- If the gate output changes to x, the minimum of the three delays is considered.
- If only one delay is specified, it is used for all delays.
- If two values are specified, they are considered as rise, and fall delays.
- If three values are specified, they are considered as rise, fall, and turn-off delays.
- The default value of all delays is zero.

```
and #(5) and_1 (out, in0, in1);
// All delay values are 5 time units.

nand #(3,4,5) nand_1 (out, in0, in1);
// rise delay = 3, fall delay = 4, and turn-off delay = 5.

or #(3,4) or_1 (out, in0, in1);
// rise delay = 3, fall delay = 4, and turn-off delay = min(3,4) = 3.
```

There is another way of specifying delay times in verilog, Min:Typ:Max values for each delay. This helps designer to have a much better real time experience of design simulation, as in real time logic circuits the delays are not constant. The user can choose one of the delay values using +maxdelays, +typdelays, and +mindelays at run time. The typical value is the default value.

```
and #(4:5:6) and_1 (out, in0, in1);
// For all delay values: Min=4, Typ=5, Max=6.

nand #(3:4:5,4:5:6,5:6:7) nand_1 (out, in0, in1);
// rise delay: Min=3, Typ=4, Max=5, fall delay: Min=4, Typ=5, Max=6, turn-off delay: Min=5, Typ=6, Max=7.
```

In the above example, if the designer chooses typical values, then rise delay = 4, fall delay = 5, turn-off delay = 6.

Examples:

1. Gate level modeling of a 4x1 multiplexer.

The gate-level circuit diagram of 4x1 mux is shown below. It is used to write a module for 4x1 mux.

```
module 4x1_mux (out, in0, in1, in2, in3, s0, s1);

// port declarations
output out; // Output port.
input in0, in1, in2. in3; // Input ports.
input s0, s1; // Input ports: select lines.

// intermediate wires
wire inv0, inv1; // Inverter outputs.
wire a0, a1, a2, a3; // AND gates outputs.

// Inverters.
not not_0 (inv0, s0);
not not_1 (inv1, s1);

// 3-input AND gates.
and and_0 (a0, in0, inv0, inv1);
and and_1 (a1, in1, inv0, s1);
and and_2 (a2, in2, s0, inv1);
and and_3 (a3, in3, s0, s1);

// 4-input OR gate.
or or_0 (out, a0, a1, a2, a3);
```

```
endmodule
```

2. Implementation of a full adder using half adders.

Half adder:

```
module half_adder (sum, carry, in0, in1);

output sum, carry;
input in0, in1;

// 2-input XOR gate.
xor xor_1 (sum, in0, in1);

// 2-input AND gate.
and and_1 (carry, in0, in1);

endmodule
```

Full adder:

```
module full_adder (sum, c_out, ino, in1, c_in);

output sum, c_out;
input in0, in1, c_in;

wire s0, c0, c1;

// Half adder : port connecting by order.
half_adder ha_0 (s0, c0, in0, in1);

// Half adder : port connecting by name.
half_adder ha_1 (.sum(sum),
            .in0(s0),
            .in1(c_in),
            .carry(c1));
```

```
// 2-input XOR gate, to get c_out.
xor xor_1 (c_out, c0, c1);

endmodule
```

Introduction

Dataflow modeling is a higher level of abstraction. The designer no need have any knowledge of logic circuit. He should be aware of data flow of the design. The gate level modeling becomes very complex for a VLSI circuit. Hence dataflow modeling became a very important way of implementing the design.
In dataflow modeling most of the design is implemented using continuous assignments, which are used to drive a value onto a net. The continuous assignments are made using the keyword *assign*.

The *assign* statement

The *assign* statement is used to make continuous assignment in the dataflow modeling. The *assign* statement usage is given below:

```
assign out = in0 + in1; // in0 + in1 is evaluated and then assigned to out.
```

Note:

- The LHS of assign statement must always be a scalar or vector net or a concatenation. It cannot be a register.
- Continuous statements are always active statements.
- Registers or nets or function calls can come in the RHS of the assignment.
- The RHS expression is evaluated whenever one of its operands changes. Then the result is assigned to the LHS.
- Delays can be specified.

Examples:

```
assign out[3:0] = in0[3:0] & in1[3:0];

assign {o3, o2, o1, o0} = in0[3:0] | {in1[2:0],in2}; // Use of concatenation.
```

Implicit Net Declaration:

```
wire in0, in1;
assign out = in0 ^ in1;
```

In the above example out is undeclared, but verilog makes an implicit net declaration for out.

Implicit Continuous Assignment:

```
wire out = in0 ^ in1;
```

The above line is the implicit continuous assignment. It is same as,

```
wire out;
assign out = in0 ^ in1;
```

Delays

There are three types of delays associated with dataflow modeling. They are: Normal/regular assignment delay, implicit continuous assignment delay and net declaration delay.

Normal/regular assignment delay:

```
assign #10 out = in0 | in1;
```

If there is any change in the operands in the RHS, then RHS expression will be evaluated after 10 units of time. Lets say that at time t, if there is change in one of the operands in the above example, then the expression is calculated at t+10 units of time. The value of RHS operands present at time t+10 is used to evaluate the expression.

Implicit continuous assignment delay:

```
wire #10 out = in0 ^ in1;
```

is same as

```
wire out;
assign 10 out = in0 ^ in1;
```

Net declaration delay:

```
wire #10 out;
assign out = in;
```

is same as

```
wire out;
assign #10 out = in;
```

Examples

1. Implementation of a 2x4 decoder.

```
module decoder_2x4 (out, in0, in1);

output out[0:3];
input in0, in1;

// Data flow modeling uses logic operators.
assign out[0:3] = { ~in0 & ~in1, in0 & ~in1,
                  ~in0 & in1, in0 & in1 };

endmodule
```

2. Implementation of a 4x1 multiplexer.

```
module mux_4x1 (out, in0, in1, in2, in3, s0, s1);

output out;
input in0, in1, in2, in3;
input s0, s1;

assign out = (~s0 & ~s1 & in0)|(s0 & ~s1 & in1)|
             (~s0 & s1 & in2)|(s0 & s1 & in0);

endmodule
```

3. Implementation of a 8x1 multiplexer using 4x1 multiplexers.

```
module mux_8x1 (out, in, sel);

output out;
input [7:0] in;
input [2:0] sel;

wire m1, m2;

// Instances of 4x1 multiplexers.
mux_4x1 mux_1 (m1, in[0], in[1], in[2],
               in[3], sel[0], sel[1]);
```

```
mux_4x1 mux_2 (m2, in[4], in[5], in[6],
               in[7], sel[0], sel[1]);

assign out = (~sel[2] & m1)|(sel[2] & m2);

endmodule
```

4. Implementation of a Full adder.

```
module full_adder (sum, c_out, in0, in1, c_in);

output sum, c_out;
input in0, in1, c_in;

assign { c_out, sum } = in0 + in1 + c_in;

endmodule
```

Introduction

Behavioral modeling is the highest level of abstraction in the Verilog HDL. The other modeling techniques are relatively detailed. They require some knowledge of how hardware, or hardware signals work. The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that designer needs is the algorithm of the design, which is the basic information for any design.

Most of the behavioral modeling is done using two important constructs: initial and always. All the other behavioral statements appear only inside these two structured procedure constructs.

The initial Construct

The statements which come under the *initial* construct constitute the initial block. The initial block is executed only once in the simulation, at time 0. If there is more than one initial block. Then all the initial blocks are executed concurrently. The initial construct is used as follows:

```
initial
begin
reset = 1'b0;
clk = 1'b1;
end
```

or

```
initial
clk = 1'b1;
```

In the first initial block there are more than one statements hence they are written between begin and end. If there is only one statement then there is no need to put begin and end.

The always Construct

The statements which come under the *always* construct constitute the always block. The always block starts at time 0, and keeps on executing all the simulation time. It works like a infinite loop. It is generally used to model a functionality that is continuously repeated.

```
always
#5 clk = ~clk;

initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10 units. The initial blocks initiates the clk value to 0 at time 0. Then after every 5 units of time it toggled, hence we get a time period of 10 units. This is the way in general used to generate a clock signal for use in test benches.

```
always @(posedge clk, negedge reset)
begin
a = b + c;
   d = 1'b1;
end
```

In the above example, the always block will be executed whenever there is a positive edge in the clk signal, or there is negative edge in the reset signal. This type of always is generally used in implement a FSM, which has a reset signal.

```
always @(b,c,d)
begin
   a = ( b + c )*d;
   e = b | c;
end
```

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the *sensitivity list.*

In the Verilog 2000, we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks is used for implementing the combination logic.

Procedural Assignments

Procedural assignments are used for updating *reg*, *integer*, *time*, *real*, *realtime*, and *memory* data types. The variables will retain their values until updated by another procedural assignment. There is a significant difference between procedural assignments and continuous assignments.
Continuous assignments drive nets and are evaluated and updated whenever an input operand changes value. Where as procedural assignments update the value of variables under the control of the procedural flow constructs that surround them.

The LHS of a procedural assignment could be:

- *reg*, *integer*, *real*, *realtime*, or *time* data type.
- Bit-select of a *reg*, *integer*, or *time* data type, rest of the bits are untouched.
- Part-select of a *reg*, *integer*, or *time* data type, rest of the bits are untouched.
- Memory word.
- Concatenation of any of the previous four forms can be specified.

When the RHS evaluates to fewer bits than the LHS, then if the right-hand side is signed, it will be sign-extended to the size of the left-hand side.

There are two types of procedural assignments: blocking and non-blocking assignments.

Blocking assignments: A blocking assignment statements are executed in the order they are specified in a sequential block. The execution of next statement begin only after the completion of the present blocking assignments. A blocking assignment will not block the execution of the next statement in a parallel block. The blocking assignments are made using the operator =.

```
initial
begin
   a = 1;
   b = #5 2;
   c = #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 7.

Non-blocking assignments: The nonblocking assignment allows assignment scheduling without blocking the procedural flow. The nonblocking assignment statement can be used whenever several variable assignments within the same time step can be made without regard to order or dependence upon each other. Non-blocking assignments are made using the operator <=.
Note: <= is same for less than or equal to operator, so whenever it appears in a expression it is considered to be comparison operator and not as non-blocking assignment.

```
initial
begin
    a <= 1;
    b <= #5 2;
    c <= #2 3;
end
```

In the above example, a is assigned value 1 at time 0, and b is assigned value 2 at time 5, and c is assigned value 3 at time 2 (because all the statements execution starts at time 0, as they are non-blocking assignments.

Block Statements

Block statements are used to group two or more statements together, so that they act as one statement. There are two types of blocks:

- Sequential block.
- Parallel block.

Sequential block: The sequential block is defined using the keywords *begin* and *end*. The procedural statements in sequential block will be executed sequentially in the given order. In sequential block delay values for each statement shall be treated relative to the simulation time of the execution of the previous statement. The control will pass out of the block after the execution of last statement.

Parallel block: The parallel block is defined using the keywords *fork* and *join*. The procedural statements in parallel block will be executed concurrently. In parallel block delay values for each statement are considered to be relative to the simulation time of entering the block. The delay control can be used to provide time-ordering for procedural assignments. The control shall pass out of the block after the execution of the last time-ordered statement.

Note that blocks can be nested. The sequential and parallel blocks can be mixed.

Block names: All the blocks can be named, by adding *: block_name* after the keyword *begin* or *fork*. The advantages of naming a block are:

- It allows to declare local variables, which can be accessed by using hierarchical name referencing.
- They can be disabled using the *disable* statement (*disable block_name;*).

Conditional (if-else) Statement

The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords *if* and *else* are used to make conditional statement. The conditional statement can appear in the following forms.

```
if ( condition_1 )
    statement_1;
```

```
if ( condition_2 )
    statement_2;
else
    statement_3;

if ( condition_3 )
    statement_4;
else if ( condition_4 )
    statement_5;
else
    statement_6;

if ( condition_5 )
begin
    statement_7;
    statement_8;
end
else
begin
    statement_9;
    statement_10;
end
```

Conditional (if-else) statement usage is similar to that if-else statement of C programming language, except that parenthesis are replaced by *begin* and *end.*

Case Statement

The case statement is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords *case* and *endcase* are used to make a case statement. The case statement syntax is as follows.

```
case (expression)
    case_item_1: statement_1;
    case_item_2: statement_2;
    case_item_3: statement_3;
    ...
    ...
    default: default_statement;
endcase
```

If there are multiple statements under a single match, then they are grouped using begin, and end keywords. The default item is optional.

Case statement with don't cares: casez and casex

casez treats high-impedance values (z) as don't cares. *casex* treats both high-impedance (z) and unknown (x) values as don't cares. Don't-care values (z values for *casez*, z and x values for *casex*) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't cares are represented using the ? mark.

Loop Statements

There are four types of looping statements in Verilog:

- forever
- repeat
- while
- for

Forever Loop

Forever loop is defined using the keyword forever, which Continuously executes a statement. It terminates when the system task $finish is called. A forever loop can also be ended by using the disable statement.

```
initial
begin
   clk = 1'b0;
   forever #5 clk = ~clk;
end
```

In the above example, a clock signal with time period 10 units of time is obtained.

Repeat Loop

Repeat loop is defined using the keyword repeat. The repeat loop block continuously executes the block for a given number of times. The number of times the loop executes can be mention using a constant or an expression. The expression is calculated only once, before the start of loop and not during the execution of the loop. If the expression value turns out to be z or x, then it is treated as zero, and hence loop block is not executed at all.

```
initial
begin
   a = 10;
   b = 5;
```

```
    b <= #10 10;
    i = 0;
    repeat(a*b)
    begin
        $display("repeat in progress");
        #1 i = i + 1;
    end
end
```

In the above example the loop block is executed only 50 times, and not 100 times. It calculates (a*b) at the beginning, and uses that value only.

While Loop

The while loop is defined using the keyword while. The while loop contains an expression. The loop continues until the expression is true. It terminates when the expression is false. If the calculated value of expression is z or x, it is treated as a false. The value of expression is calculated each time before starting the loop. All the statements (if more than one) are mentioned in blocks which begins and ends with keyword begin and end keywords.

```
initial
begin
    a = 20;
    i = 0;
    while (i < a)
    begin
    $display("%d",i);
    i = i + 1;
    a = a - 1;
    end
end
```

In the above example the loop executes for 10 times. (observe that a is decrementing by one and i is incrementing by one, so loop terminated when both i and a become 10).

For Loop

The For loop is defined using the keyword for. The execution of for loop block is controlled by a three step process, as follows:

1. Executes an assignment, normally used to initialize a variable that controls the number of times the for block is executed.
2. Evaluates an expression, if the result is false or z or x, the for-loop shall terminate, and if it is true, the for-loop shall execute its block.

3. Executes an assignment normally used to modify the value of the loop-control variable and then repeats with second step.

Note that the first step is executed only once.

```
initial
begin
    a = 20;
    for (i = 0; i < a; i = i + 1, a = a - 1)
    $display("%d",i);
end
```

The above example produces the same result as the example used to illustrate the functionality of the while loop.

Examples

1. Implementation of a 4x1 multiplexer.

```
module 4x1_mux (out, in0, in1, in2, in3, s0, s1);

output out;

// out is declared as reg, as default is wire

reg out;

// out is declared as reg, because we will
// do a procedural assignment to it.

input in0, in1, in2, in3, s0, s1;

// always @(*) is equivalent to
// always @( in0, in1, in2, in3, s0, s1 )

always @(*)
begin
  case ({s1,s0})
      2'b00: out = in0;
      2'b01: out = in1;
      2'b10: out = in2;
      2'b11: out = in3;
      default: out = 1'bx;
  endcase
```

```
end

endmodule
```

2. Implementation of a full adder.

```
module full_adder (sum, c_out, in0, in1, c_in);

output sum, c_out;
reg sum, c_out

input in0, in1, c_in;

always @(*)
  {c_out, sum} = in0 + in1 + c_in;

endmodule
```

3. Implementation of a 8-bit binary counter.

```
module ( count, reset, clk );

output [7:0] count;
reg [7:0] count;

input reset, clk;

// consider reset as active low signal

always @( posedge clk, negedge reset)
begin
  if(reset == 1'b0)
      count <= 8'h00;
  else
      count <= count + 8'h01;
end

endmodule
```

Implementation of a 8-bit counter is a very good example, which explains the advantage of behavioral modeling. Just imagine how difficult

it will be implementing a 8-bit counter using gate-level modeling.
In the above example the incrementation occurs on every positive edge of the clock. When count becomes 8'hFF, the next increment will make it 8'h00, hence there is no need of any modulus operator. Reset signal is active low.

Introduction

Tasks and functions are introduced in the verilog, to provide the ability to execute common procedures from different places in a description. This helps the designer to break up large behavioral designs into smaller pieces. The designer has to abstract the similar pieces in the description and replace them either functions or tasks. This also improves the readability of the code, and hence easier to debug. Tasks and functions must be defined in a module and are local to the module. Tasks are used when:

- There are delay, timing, or event control constructs in the code.
- There is no input.
- There is zero output or more than one output argument.

Functions are used when:

- The code executes in zero simulation time.
- The code provides only one output(return value) and has at least one input.
- There are no delay, timing, or event control constructs.

Differences

Functions	**Tasks**
Can enable another function but not another task.	Can enable other tasks and functions.
Executes in 0 simulation time.	May execute in non-zero simulation time.
Must not contain any delay, event, or timing control statements.	May contain delay, event, or timing control statements.
Must have at least one input argument. They can have more than one input.	May have zero or more arguments of type input, output, or inout.
Functions always return a single value. They cannot have output or inout arguments.	Tasks do not return with a value, but can pass multiple values through output and inout arguments.

Tasks

There are two ways of defining a task. The first way shall begin with the keyword task, followed by the optional keyword automatic, followed by a name for the task, and ending with the keyword endtask. The keyword automatic declares an automatic task that is reentrant with all the task declarations allocated dynamically for each concurrent task entry. Task item declarations can specify the following:

- Input arguments.

- Output arguments.
- Inout arguments.
- All data types that can be declared in a procedural block

The second way shall begin with the keyword task, followed by a name for the task and a parenthesis which encloses task port list. The port list shall consist of zero or more comma separated ports. The task body shall follow and then the keyword endtask.

In both ways, the port declarations are same. Tasks without the optional keyword automatic are static tasks, with all declared items being statically allocated. These items shall be shared across all uses of the task executing concurrently. Task with the optional keyword automatic are automatic tasks. All items declared inside automatic tasks are allocated dynamically for each invocation. Automatic task items can not be accessed by hierarchical references. Automatic tasks
can be invoked through use of their hierarchical name.

Functions

Functions are mainly used to return a value, which shall be used in an expression. The functions are declared using the keyword function, and definition ends with the keyword endfunction.

If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space. The keyword automatic declares a recursive function with all the function declarations allocated dynamically for each recursive call. Automatic function items can not be accessed by hierarchical references. Automatic functions can be invoked through the use of their hierarchical name.

When a function is declared, a register with function name is declared implicitly inside Verilog HDL. The output of a function is passed back by setting the value of that register appropriately.

Examples

1. Simple task example, where task is used to get the address tag and offset of a given address.

```
module example1_task;

input addr;
wire [31:0] addr;

wire [23:0] addr_tag;
wire [7:0] offset;

task get_tag_and_offset ( addr, tag, offset);

input addr;
```

```
output tag, offset;

begin
 tag = addr[31:8];
 offset = addr[7:0];
end
endtask

always @(addr)
begin
 get_tag_and_offset (addr, addr_tag, addr_offset);
end

// other internals of module

endmodule
```

2. Task example, which uses the global variables of a module. Here task is used to do temperature conversion.

```
module example2_global;

real t1;
real t2;

// task uses the global variables of the module

task t_convert;
begin
 t2 = (9/5)*(t1+32);
end
endtask

always @(t1)
begin
 t_convert();
end

endmodule
```