

UNIT I

INTRODUCTION TO MOS TRANSISTOR

Basic MOSFET Structure

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

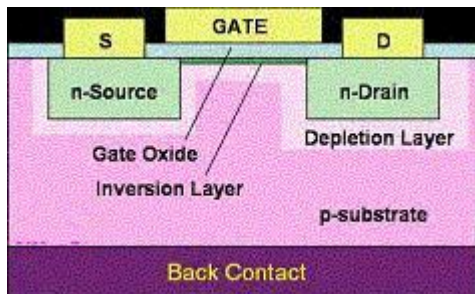


fig 1 Cross-sectional view of MOSFET

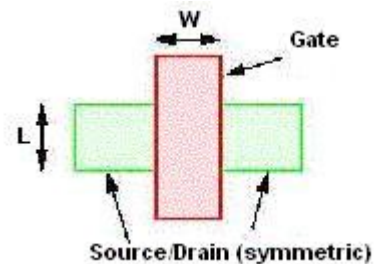


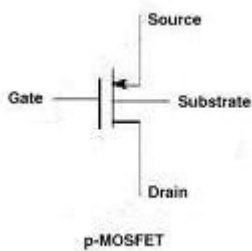
fig 2 Top/Bottom View of MOSFET An n-

MOSFET

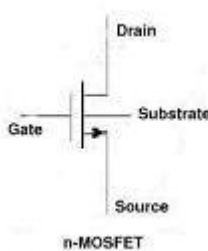
It consists of a **source** and a **drain**, two highly conducting n-type semiconductor regions which are separated from the p-type substrate by reverse-biased p-n diodes. A metal or poly crystalline gate covers the region between the source and drain, but is isolated from the semiconductor by the **gate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

Types of MOSFET

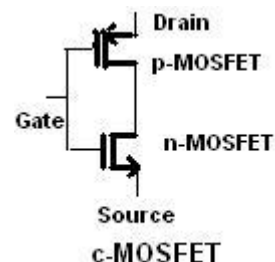
MOSFETs are divided into two types viz. **p-MOSFET** and **n-MOSFET** depending upon its type of source and drain.



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 output characteristics () and the transfer characteristics 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 :

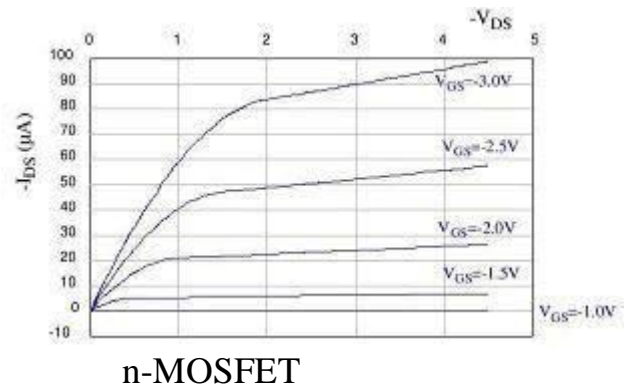
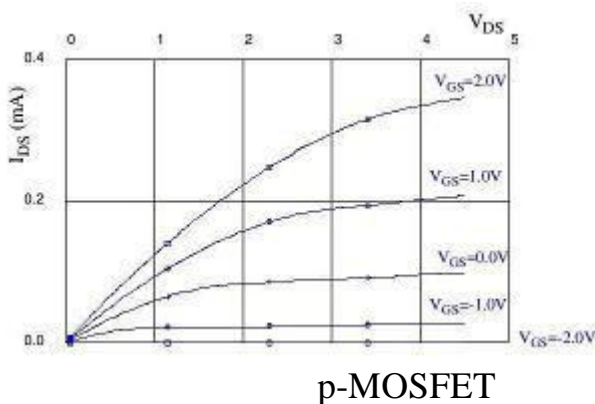
p-MOSFET:

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

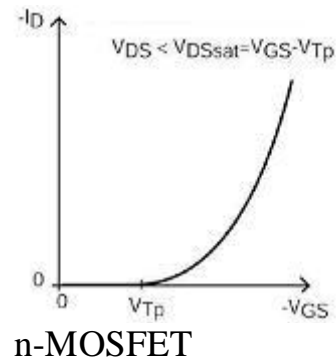
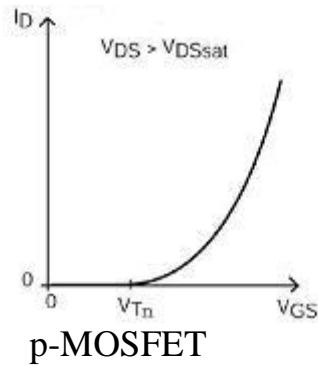
n-MOSFET:

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

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

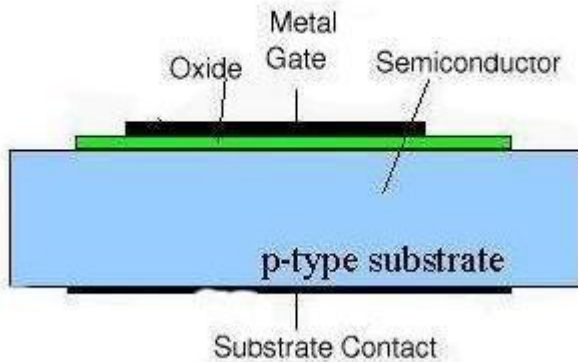


The **transfer** characteristics of both **p-MOSFET** and **n-MOSFET** are plotted for a fixed value of as shown next :

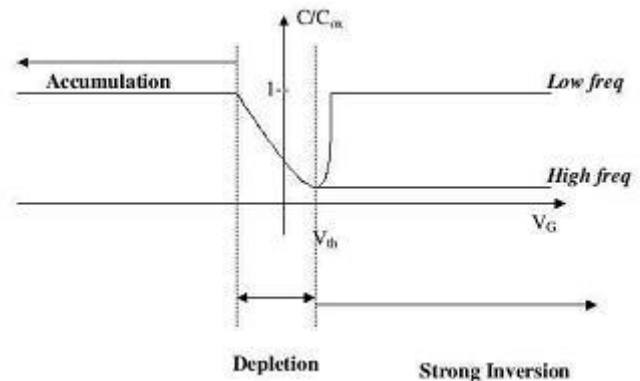


C-V Characteristics of a MOS Capacitor

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 (**V_{th}**) of MOS capacitor.



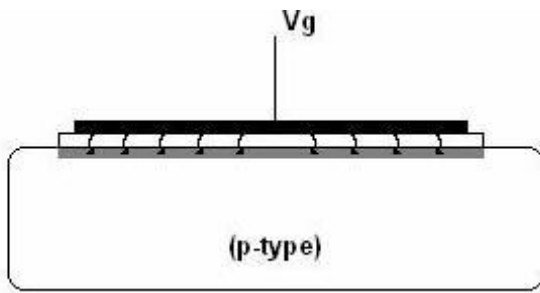
Cross section view of MOS Capacitor



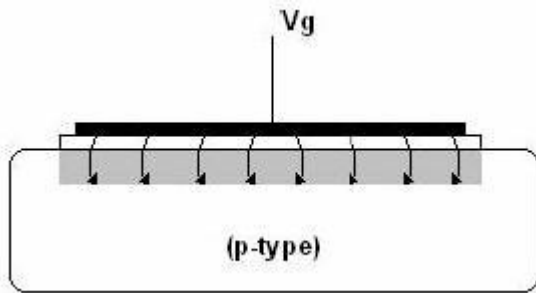
plot of MOS Capacitor

Modes of operation

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

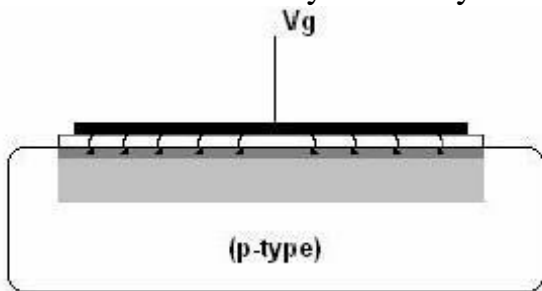


Accumulation mode (grey layer - strong hole concentration)



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, $V_g < 0$
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.
3. **Strong Inversion:** When V_g 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 - strong electron concentration, light grey - depletion region)

Threshold voltage

$$\phi_F = \frac{kT}{q} \ln\left(\frac{N_A}{n_i}\right)$$

It is that gate voltage at which the surface band bending is twice ϕ_F , Where

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

$$x_d = \sqrt{\frac{2\epsilon_s \phi_s}{qN_a}}$$

Charge in depletion region at $\phi_s = 2\phi_F$ is given by $Q_D = -qN_a x_{dmax}$ where

$$x_{dmax} = \sqrt{\frac{2\epsilon_s (2\phi_F)}{qN_a}}$$

Beyond threshold, the total charge Q_D in the semiconductor has to balance the charge on gate electrode, Q_s 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-

$$V_{GS} = V_{FB} + \phi_s + \frac{Q_s}{C_{ox}} = V_{FB} + \phi_s - \frac{(Q_i + Q_D)}{C_{ox}}$$

In case of depletion, there is no inversion layer charge, so $Q_i = 0$, i.e. gate voltage becomes

$$V_{GS} = V_{FB} + \phi_s - \frac{Q_D}{C_{ox}} = V_{FB} + \phi_s + \frac{2\sqrt{qN_a \epsilon_s \phi_F}}{C_{ox}} \text{ for } 0 \leq \phi_s \leq 2\phi_F$$

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

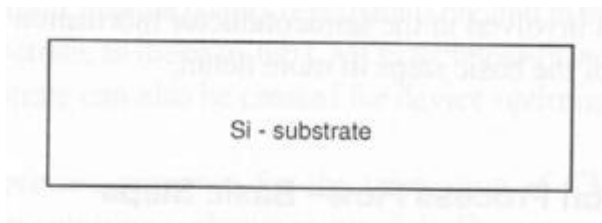
The second term in second equality of last expression states our basic assumption, namely that any change in gate voltage beyond the threshold requires a change in inversion layer charge. Also from the same expression, we obtain threshold voltage as :

$$V_T = V_{FB} + 2\phi_F + \frac{2\sqrt{qN_a \epsilon_s \phi_F}}{C_{ox}}$$

MOS Fabrication:

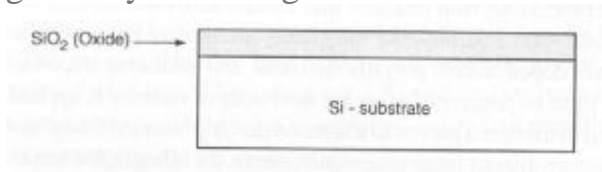
Step1:

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^{15} to 10^{16} /cm³.



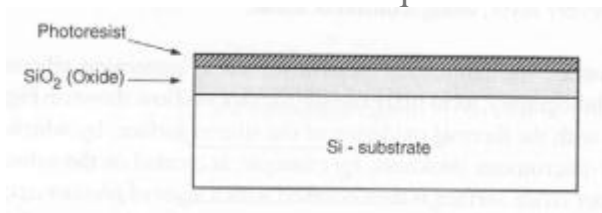
Step 2 :

A layer of silicon di oxide (SiO_2) 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.



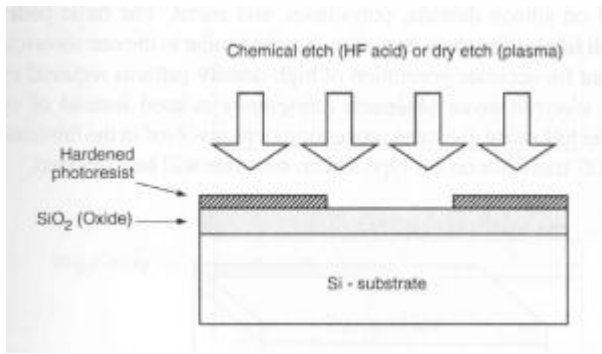
Step 4:

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.



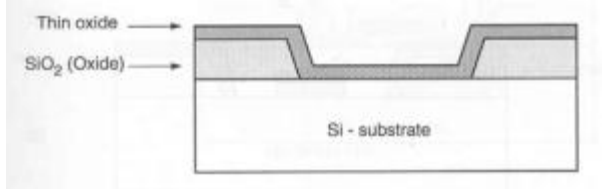
Step 5:

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:

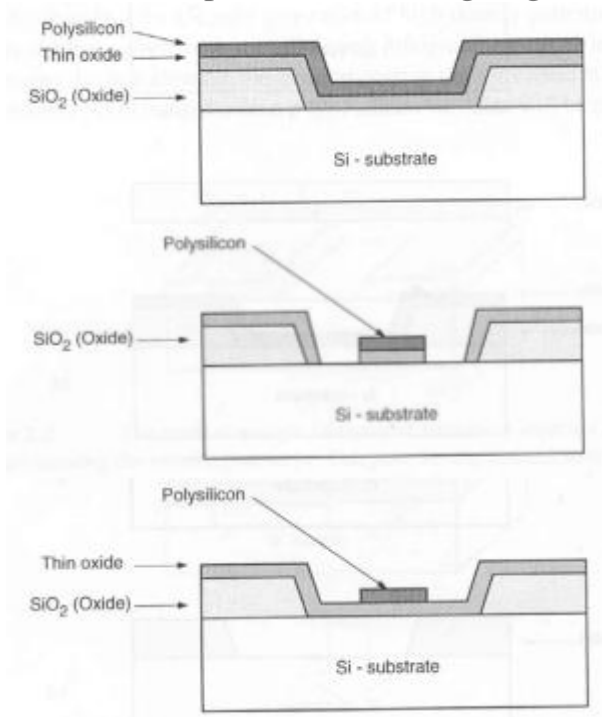
The remaining photo resist is removed and a thin layer of SiO₂ (0.1 micro m typical) is grown over the entire chip surface and then poly silicon is deposited on the top of this to form the gate structure. The polysilicon layer consists of heavily doped polysilicon deposited by chemical vapour deposition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



Step 7:

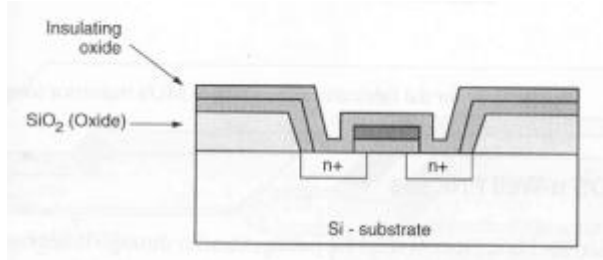
Further photo resist coating and masking allows the poly silicon to be patterned and then the thin oxide is removed to expose areas into which n-type impurities are to be diffused to form the source and drain. Diffusion is achieved by heating the wafer to a high temperature and passing a gas containing 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 (SiO_2) 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)

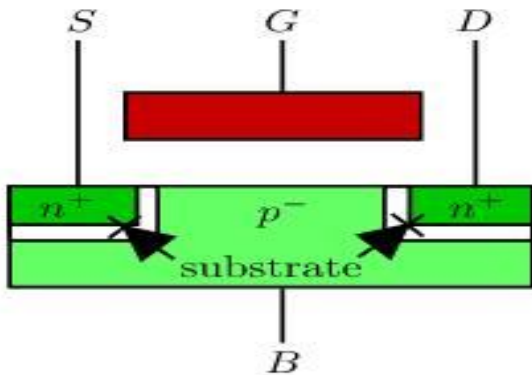


Step 9:

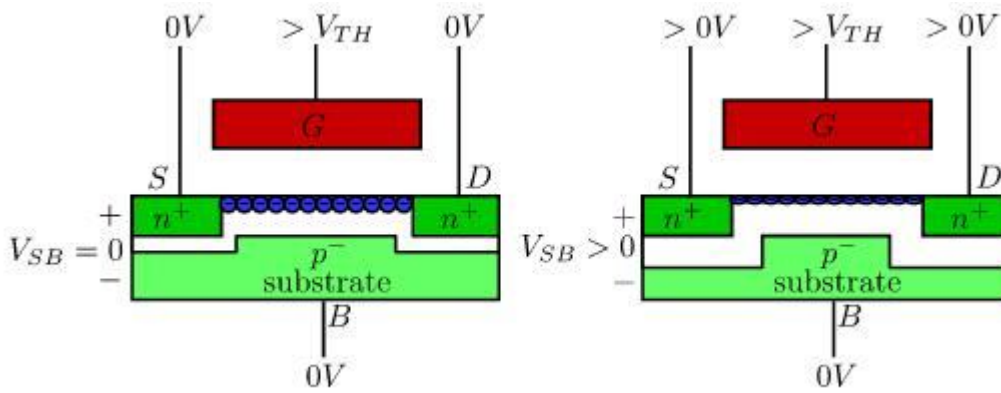
The whole chip then has metal (aluminium) deposited over its surface to a thickness typically of 1 micro m. This metal layer is then masked and etched to form the required interconnection pattern.

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 $V_S > V_B$, the depletion width of the pn junction increases. That makes it more difficult to create a channel with the same V_{GS} , effectively reducing the channel depth. In order to return to the same channel depth, V_{GS} 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 channel length (reduced by ΔL) and increased drain current. When the channel length of MOSFET is decreased and MOSFET is operated beyond channel pinch-off, the relative importance of pinchoff length ΔL with respect to physical length is increased. This effect can be included in saturation current as :

$$I'_{DSat} = \frac{I_{DSat}}{1 - \frac{\Delta L}{L}}$$
$$I_{DSat} = \mu_n \frac{W}{L} C_{ox} \frac{(V_{GS} - V_T)^2}{2} (1 + \lambda V_{DS})$$

Here λ is called channel length modulation coefficient

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 independently.

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 characteristics 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.

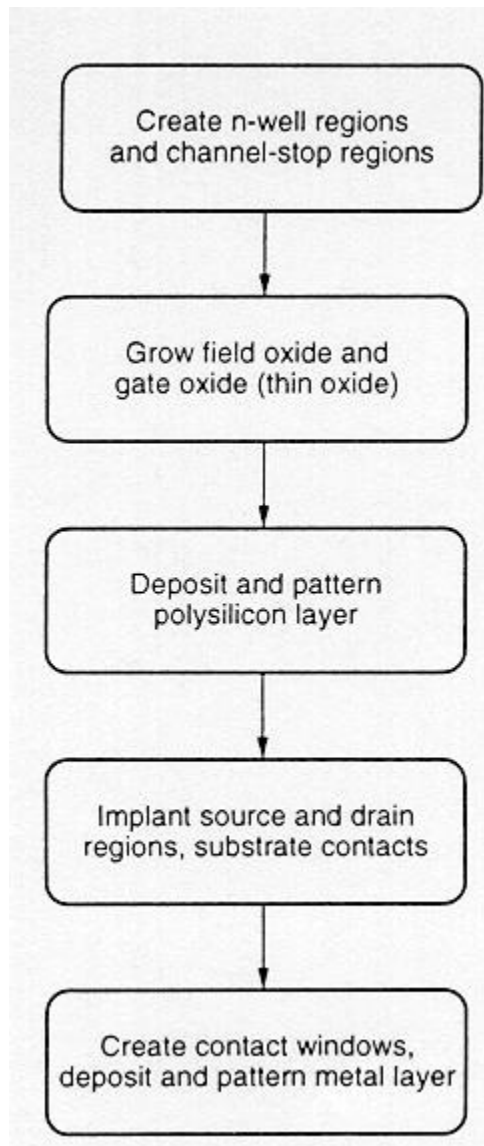
N-well Technology

In this discussion we will concentrate on the well established n-well CMOS fabrication technology, which requires that both n-channel and p-channel transistors be built on the same chip substrate. To accommodate 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:

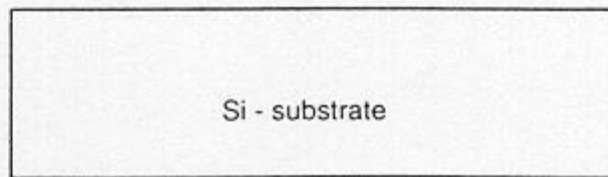
- N-well regions are created for PMOS transistors, by impurity implantation into the substrate.
- This is followed by the growth of a thick oxide in the regions surround the NMOS and PMOS active regions.
- The thin gate oxide is subsequently grown on the surface through thermal oxidation.
- After this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



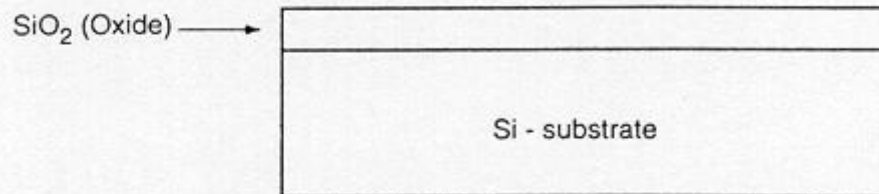
Simplified Process Sequence For Fabrication Of CMOS ICs

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.

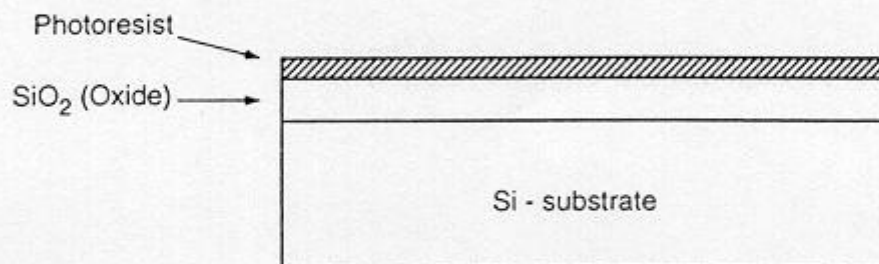
We illustrate the fabrication steps involved in patterning silicon dioxide through optical lithography, which shows the lithographic sequences.



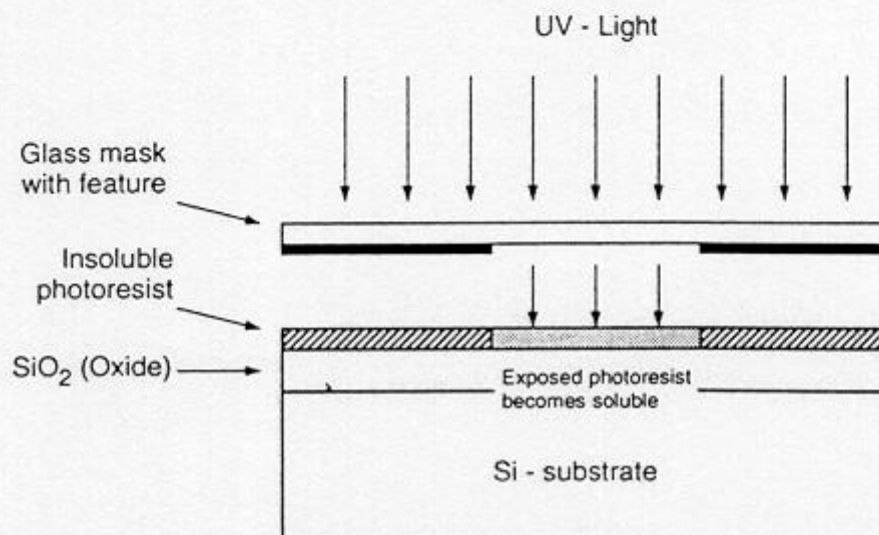
(a)



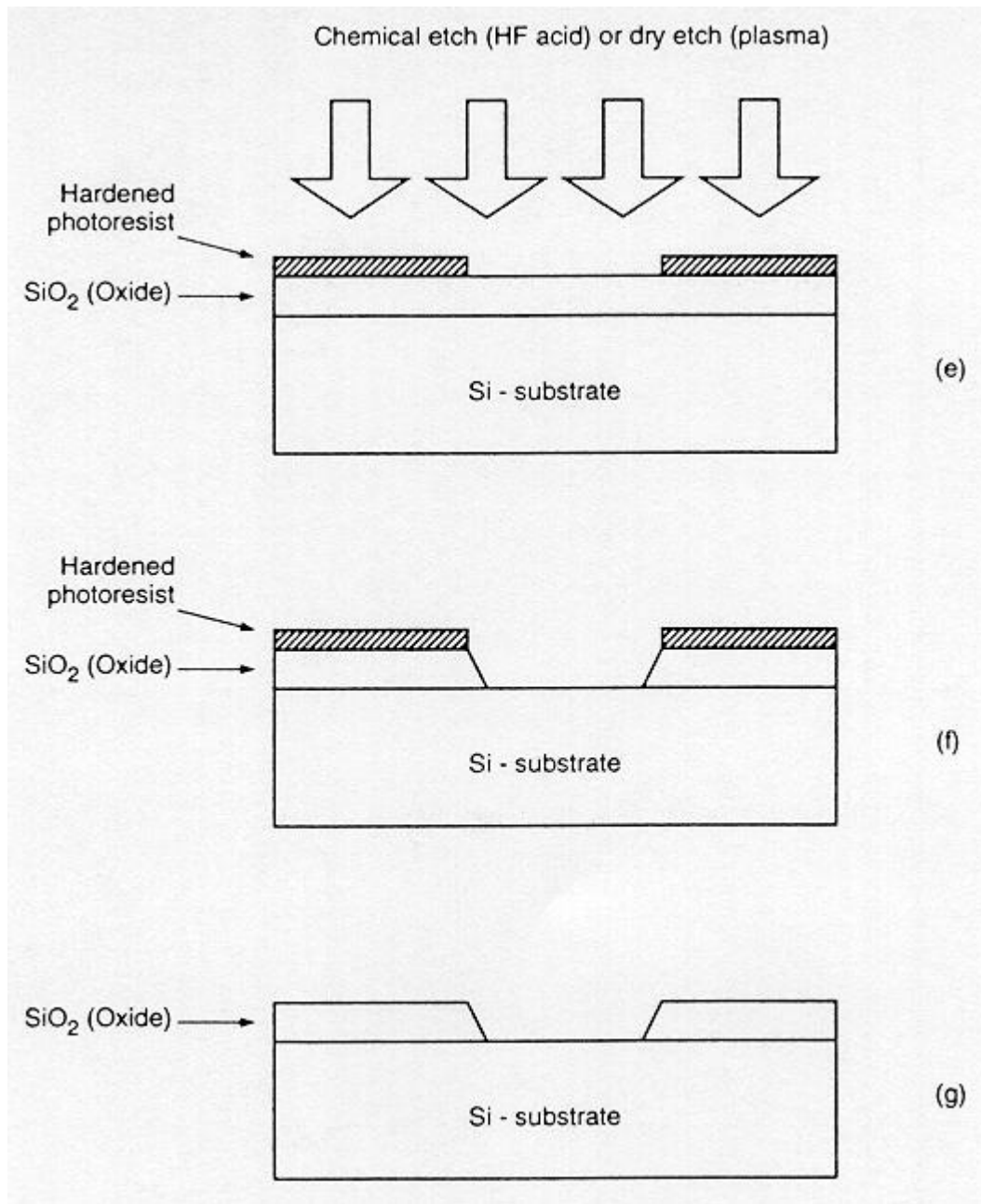
(b)



(c)



(d)



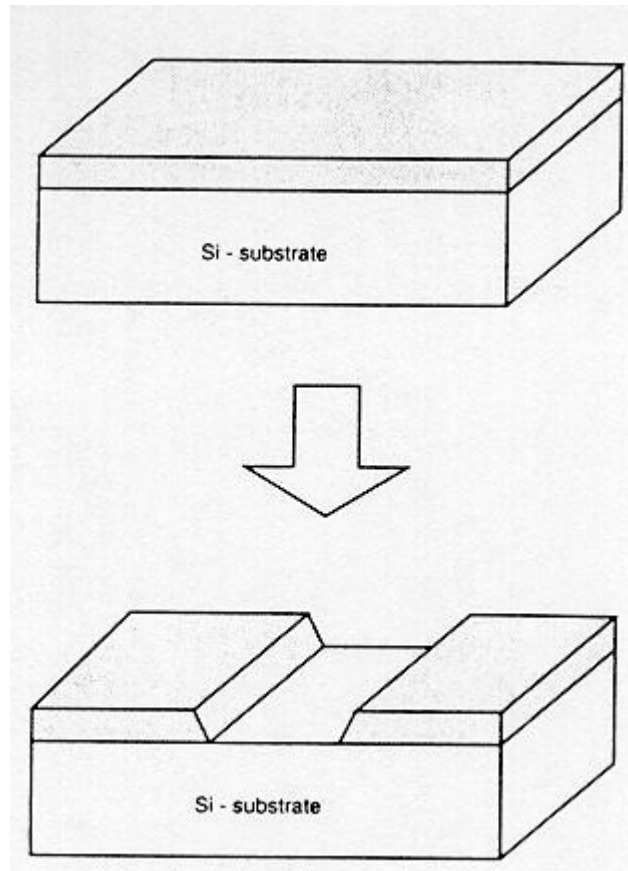
Process steps required for patterning of silicon dioxide

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, whereas 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

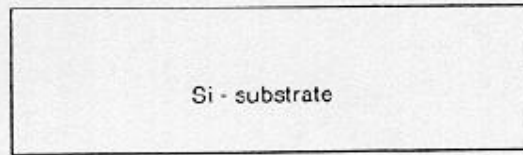
The sequence of process steps illustrated in detail actually accomplishes a single pattern transfer onto the silicon dioxide surface. The fabrication of semiconductor devices requires several such pattern transfers to be performed on silicon dioxide, polysilicon, and metal. The basic patterning process 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.

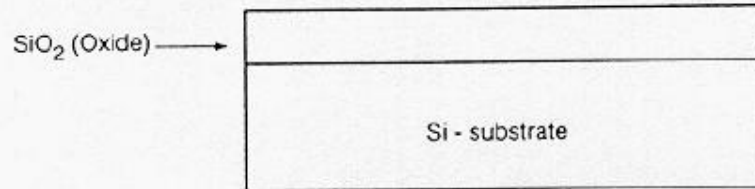
The first step of the process is the oxidation of the silicon substrate which creates a relatively thick silicon dioxide layer on the surface. This oxide layer is called field oxide. The field oxide is then selectively etched to expose the silicon surface on which the transistor will be created. After 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.

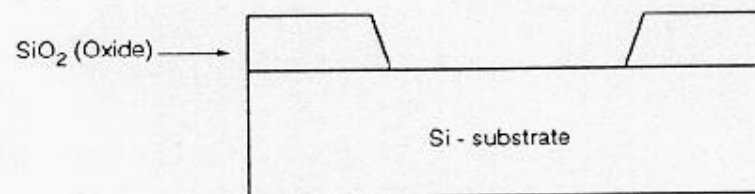
The entire surface is again covered with an insulating layer of silicon dioxide after the source and drain regions are completed. Next contact windows for the source and drain are patterned into the oxide layer. Interconnects are formed by evaporating aluminium on the surface which is followed by patterning 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.



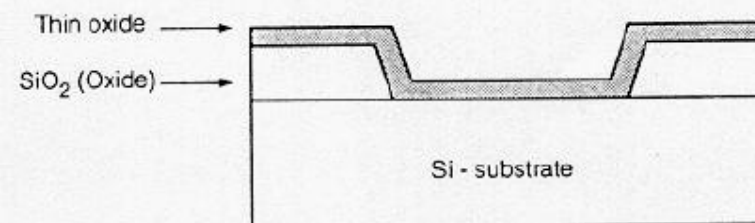
(a)



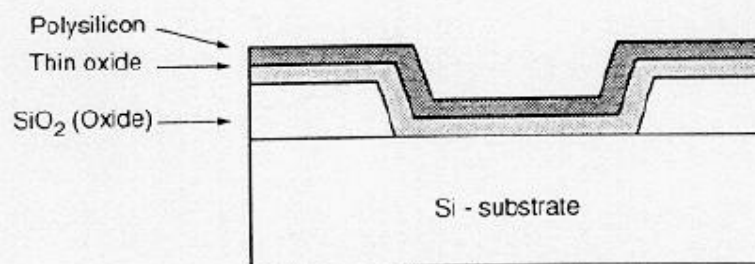
(b)



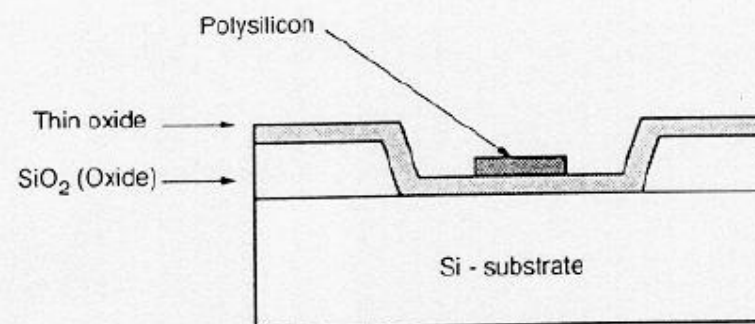
(c)



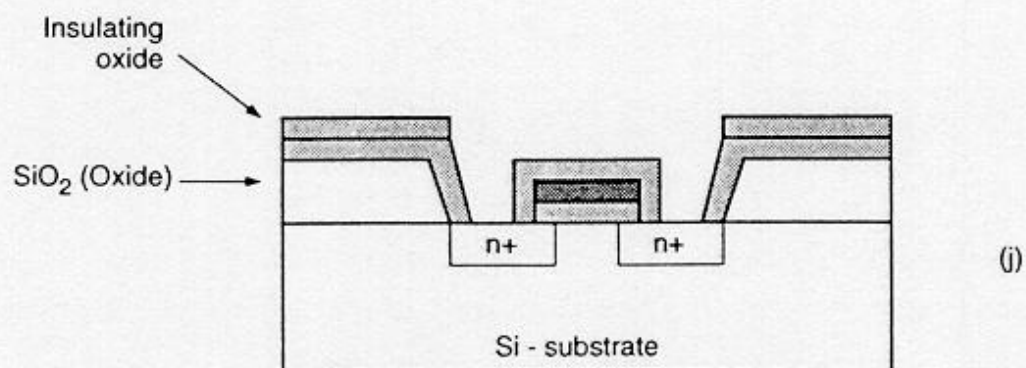
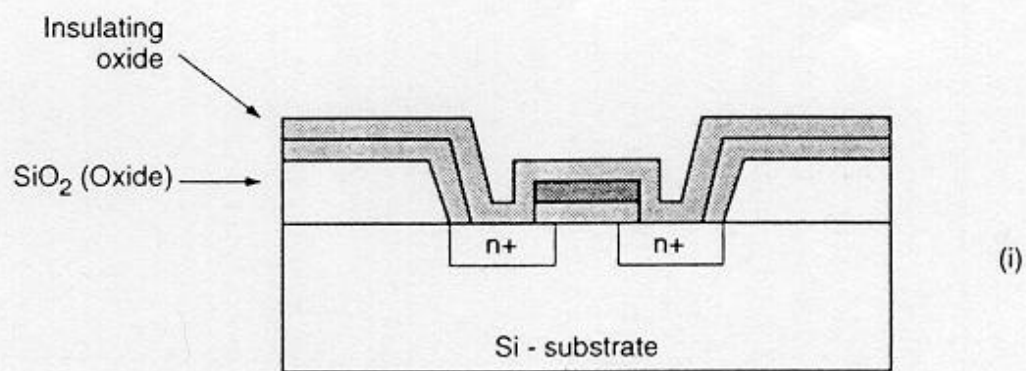
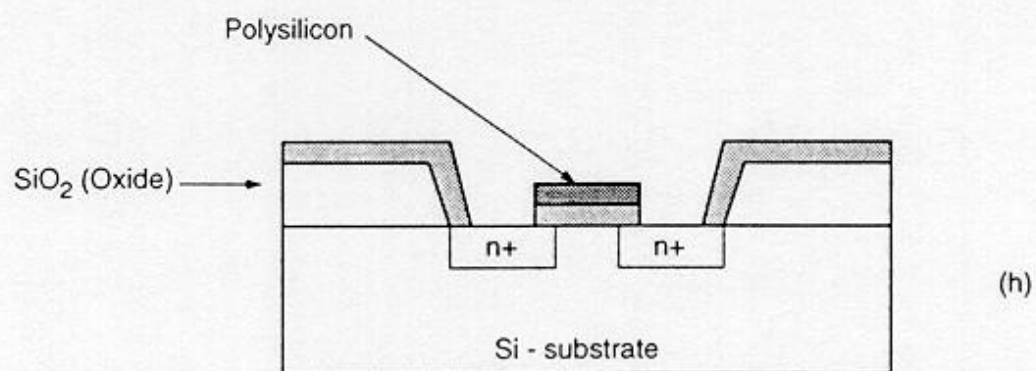
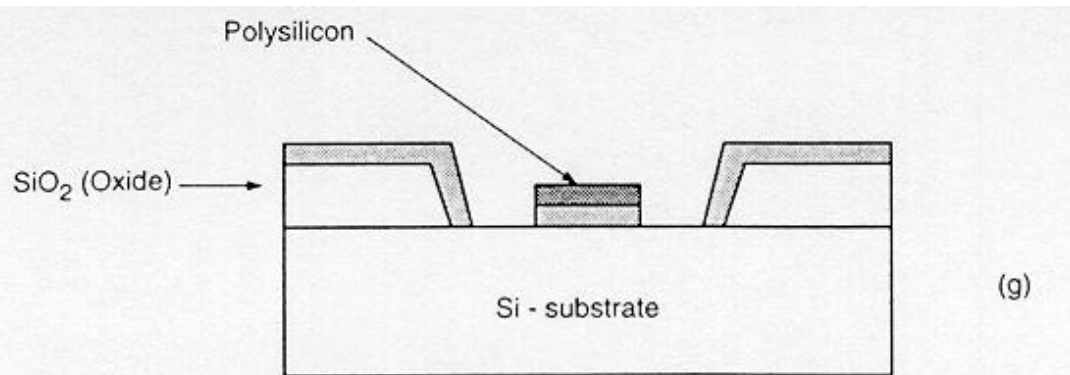
(d)

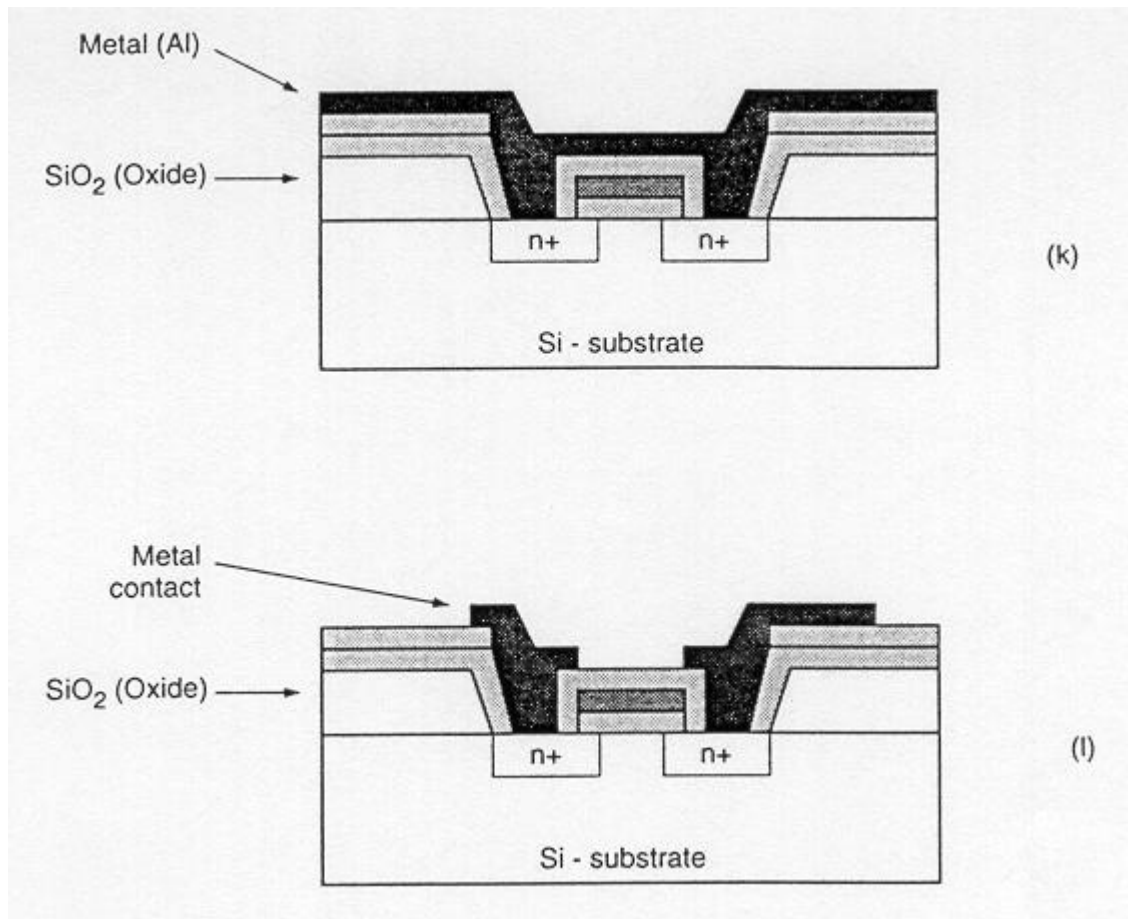


(e)



(f)

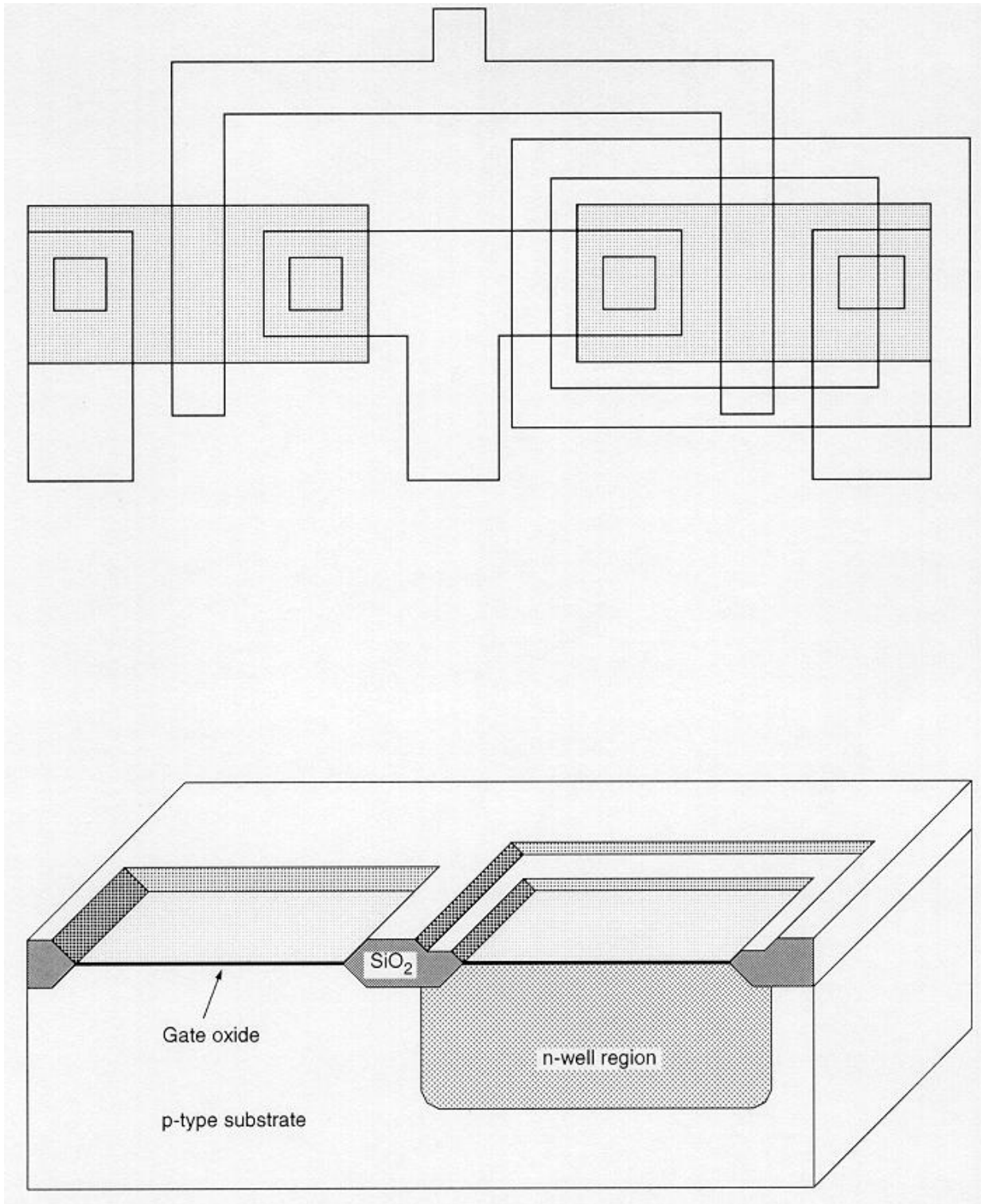




Process flow for the fabrication of an n-type MOSFET on p-type silicon

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 10^{15} 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

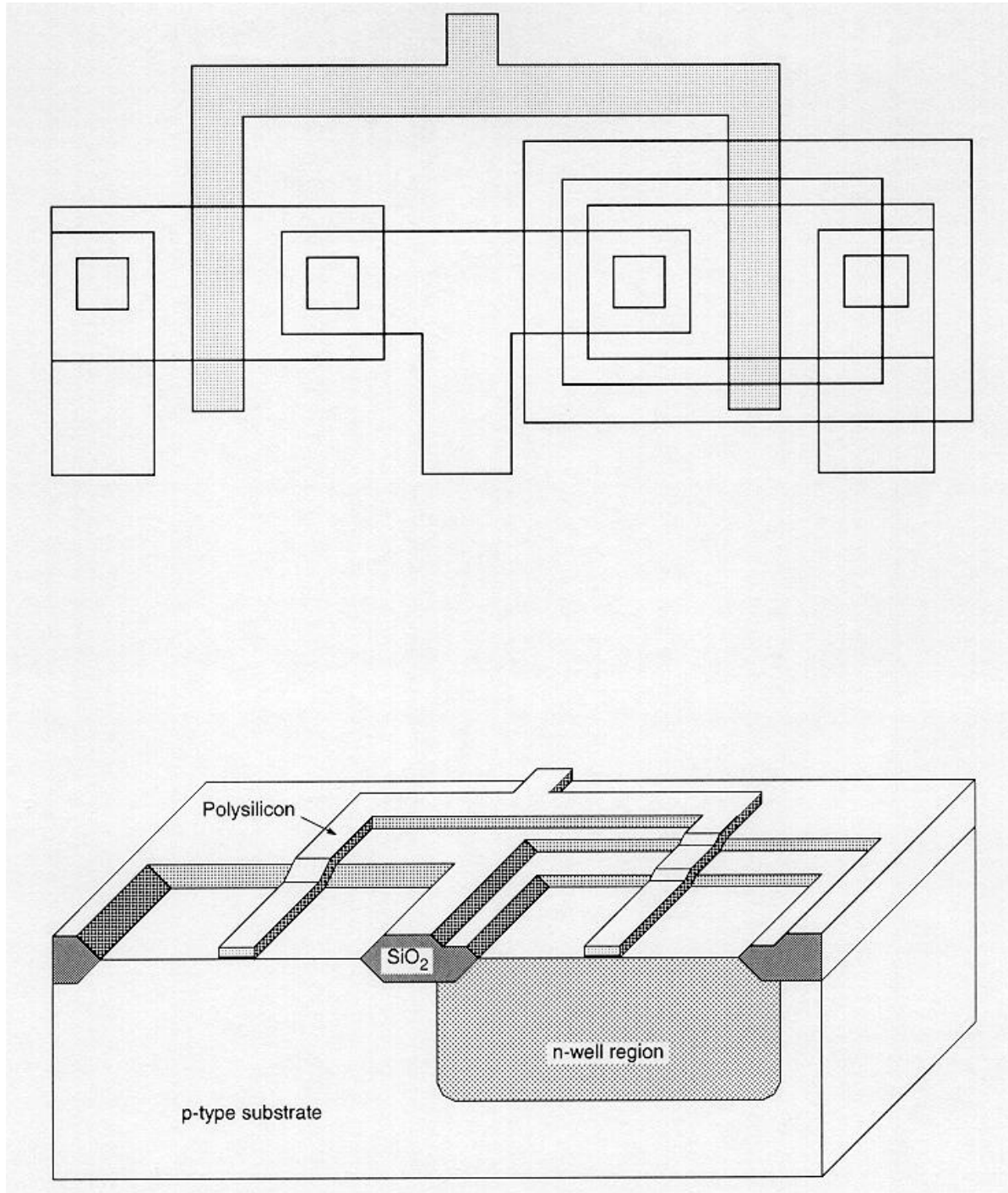


The creation of the n-well region is followed by the growth of a thick field oxide in the areas surrounding the transistor active regions, and a thin gate oxide on top of the active regions. The two most important critical fabrication parameters are the thickness and quality of the gate oxide. These 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.

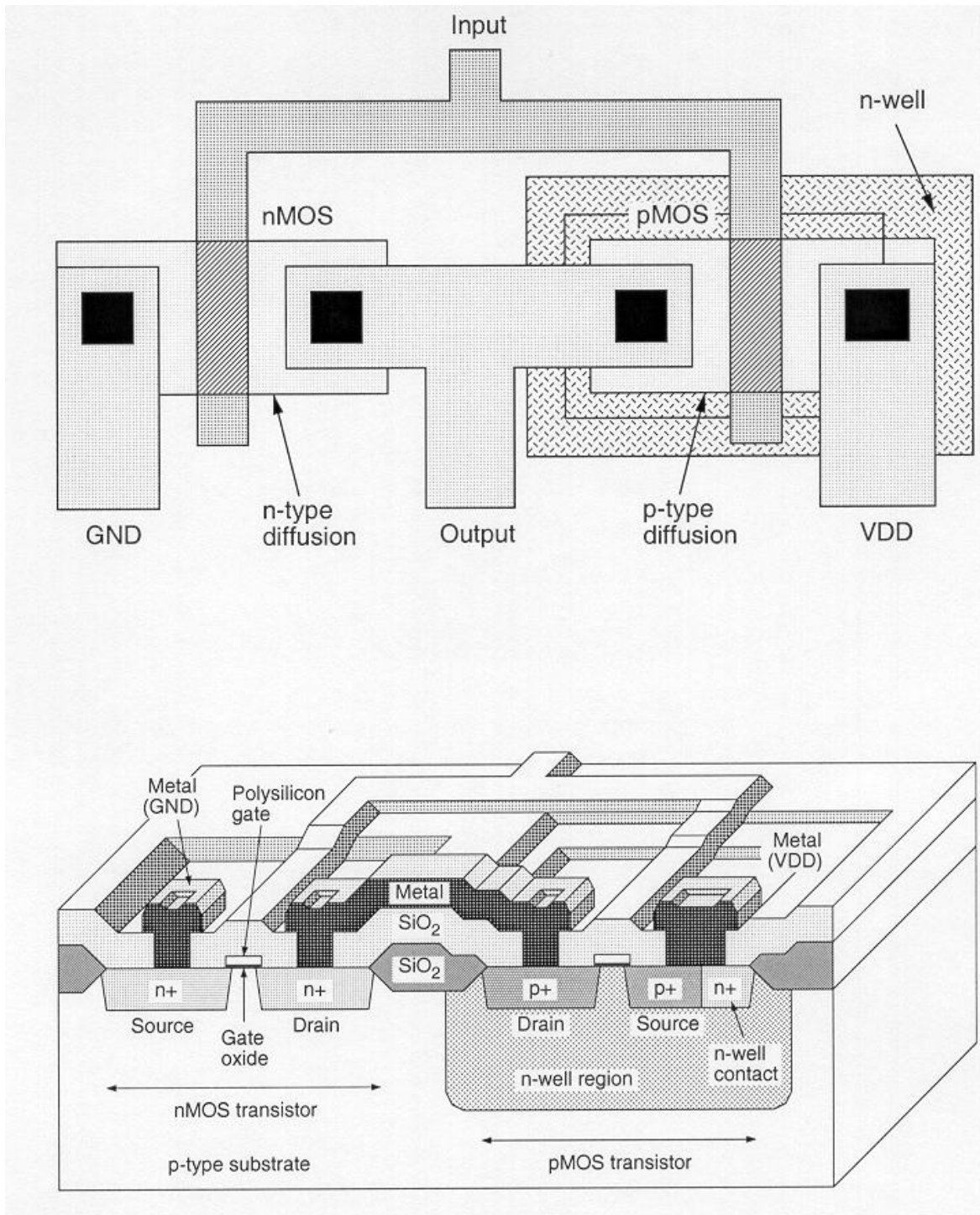


CVD is again used to deposit an insulating silicon dioxide layer over the entire wafer. After this the contacts are defined and etched away exposing the silicon or polysilicon contact windows. These contact windows are essential to complete the circuit interconnections using the metal layer, which is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



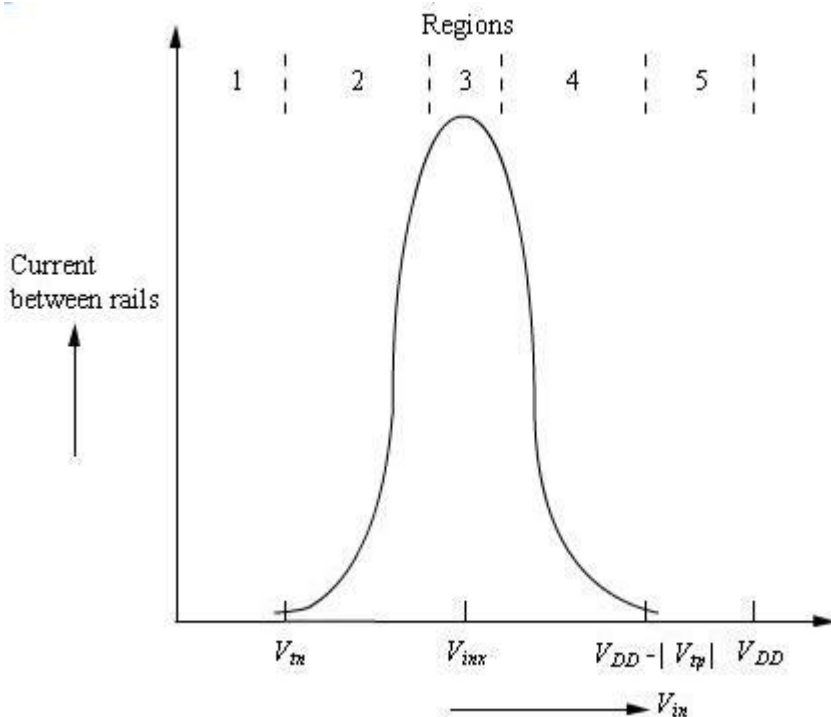
DC Characteristics of CMOS:

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

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

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 $0 \leq V_{in} < V_{tn}$. 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 $V_{tn} \leq V_{in} < V_{inv}$. 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 $V_{gs} = V_{inv}$, $V_{gd} = V_{inv} - V_{DD}$).

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

where $\beta = K \frac{W}{L}$ and $K = \frac{\epsilon_{ox} \epsilon_0 \mu}{D}$. 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, $V_{inv} < V_{in} \leq V_{DD} - |V_{tp}|$. 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, $V_{DD} - |V_{tp}| \leq V_{in} \leq V_{DD}$. 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 $V_{out} \approx 0$.

UNIT II

MOS CIRCUITS DESIGN PROCESS AND CMOS LOGIC GATES

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. 0.75 μ m) 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.

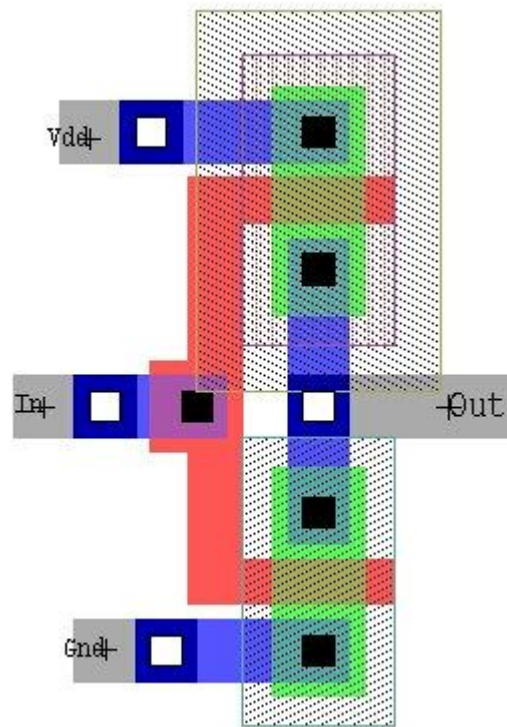
The layers for typical CMOS processes are represented in various figures in terms of:

- 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

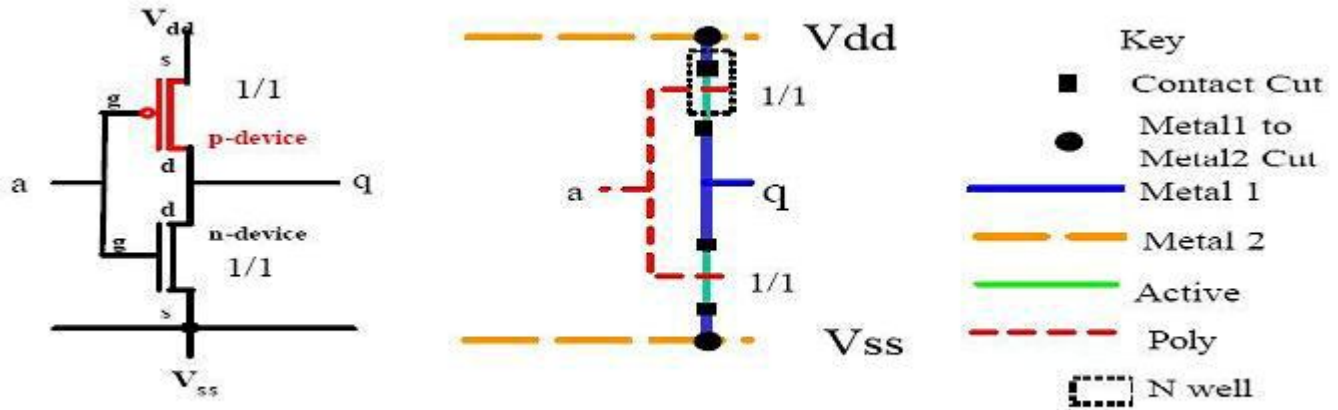
Stick Diagrams

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.

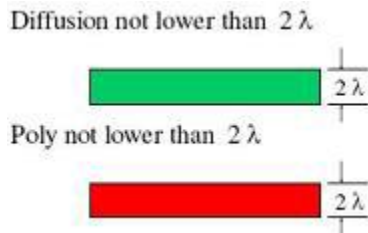
λ -based Design Rules

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

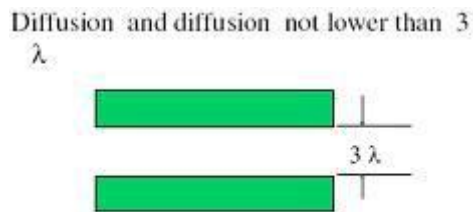
- λ is the size of a minimum feature
- All the dimensions are specified in integer multiple of λ .

- 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

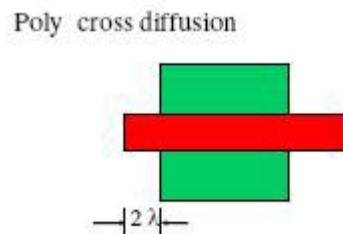
Guidelines for using λ -based Design Rules:



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

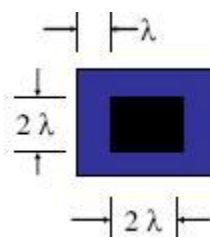


As Minimum distance between two diffusion layers 3λ



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

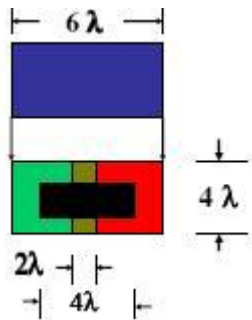


Contact window will be of 2λ by 2λ that is minimum feature size while metal deposition is of 4λ by 4λ for reliable contacts.

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.

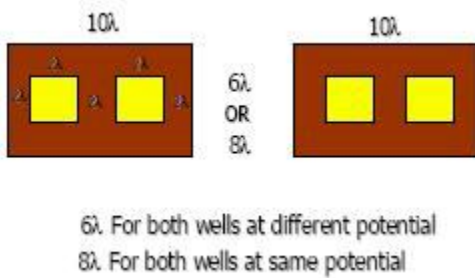


Buttering contact

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.



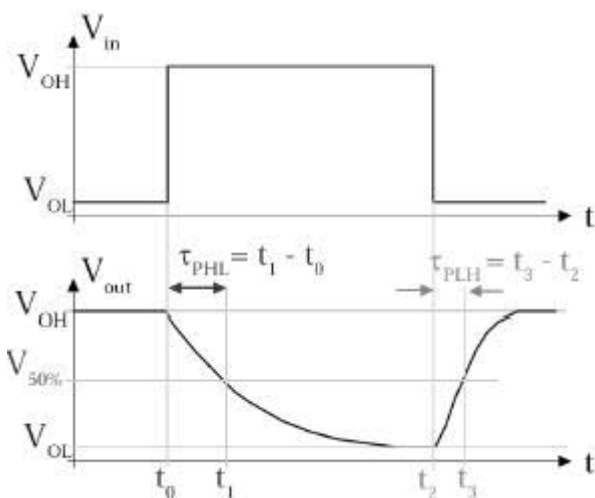
The active region has length 10λ which is distributed over the followings-

- 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, CL.
- **Rise time, t_r** : waveform to rise from 10% to 90% of its steady state value
- **Fall time t_f** : 90% to 10% of steady state value
- **Delay time, t_d** : time difference between input transition (50%) and 50% output level



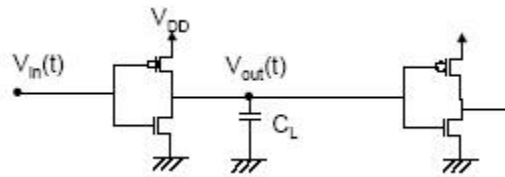
Propagation delay graph

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_{pLH} 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.

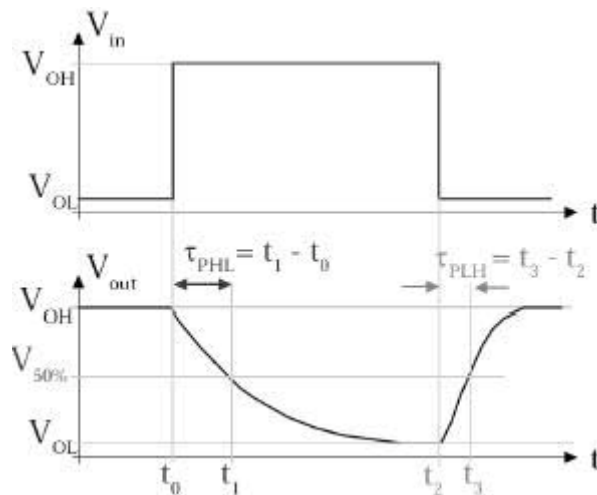


Example CMOS Inverter Circuit

$$V_{50\%} = V_{OL} + \frac{V_{OH} - V_{OL}}{2} = \frac{V_{OH} + V_{OL}}{2}$$

$$V_{90\%} = V_{OL} + 0.9(V_{OH} - V_{OL})$$

$$V_{10\%} = V_{OL} + 0.1(V_{OH} - V_{OL})$$



Propagation Delay of above MOS circuit

From figure, when $V_{in} = 0$ the capacitor C_L charges through the PMOS, and when $V_{in} = 5$ 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 (16.22) can be easily seen to be

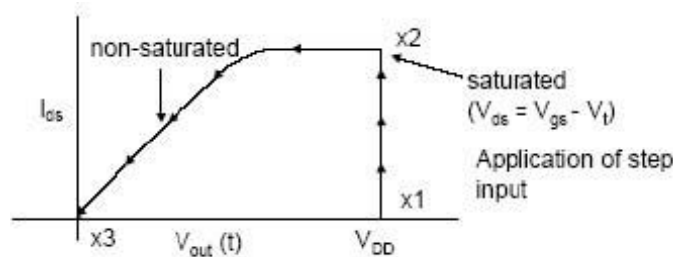
$$\tau_{PHL} = \frac{C_{Load} \Delta V_{HL}}{I_{avg,HL}} = \frac{C_{Load} (V_{OH} - V_{50\%})}{I_{avg,HL}} \quad \& \quad \tau_{PLH} = \frac{C_{Load} \Delta V_{LH}}{I_{avg,LH}} = \frac{C_{Load} (V_{50\%} - V_{OL})}{I_{avg,LH}}$$

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

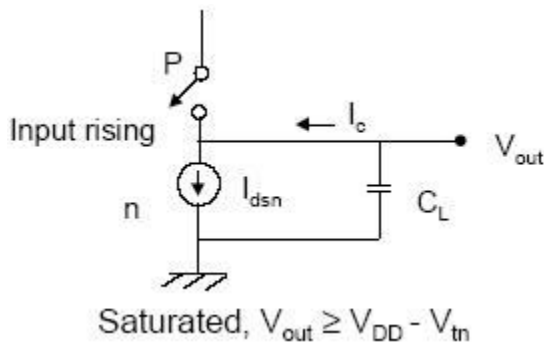


trajectory of n-transistor operating point

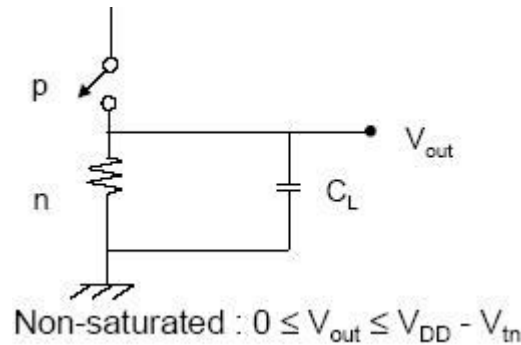
Above Figure shows the trajectory of the n-transistor operating point as the input voltage, **V_{in}(t)**, changes from **0V** to **V_{DD}**. Initially, the end-device is cutt-off and the load capacitor is charged to **V_{DD}**. This illustrated by **X1** on the characteristic curve. Application of a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

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

1. **tf1**=period during which the capacitor voltage, **V_{out}**, drops from **0.9V_{DD}** to **(V_{DD}–V_{tn})**
2. **tf2**=period during which the capacitor voltage, **V_{out}**, drops from **(V_{DD}–V_{tn})** to **0.1V_{DD}**.



Equivalent circuit for showing behavior of t_{f1}



Equivalent circuit for showing behavior of t_{f2}

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

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

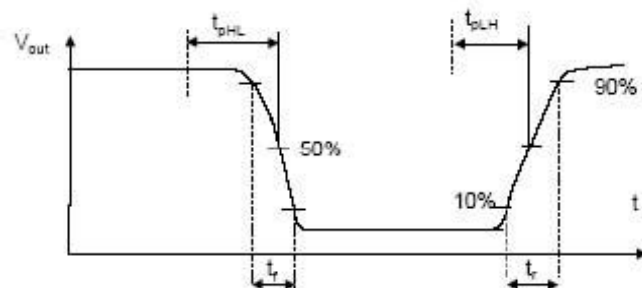
while in saturation,

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

Integrating from $t = t_1$, corresponding to $V_{out} = 0.9 V_{DD}$, 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 **(VDD-Vtn)** to **0.1VDD** can be obtained as before. In linear region,

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

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

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

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,

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

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.

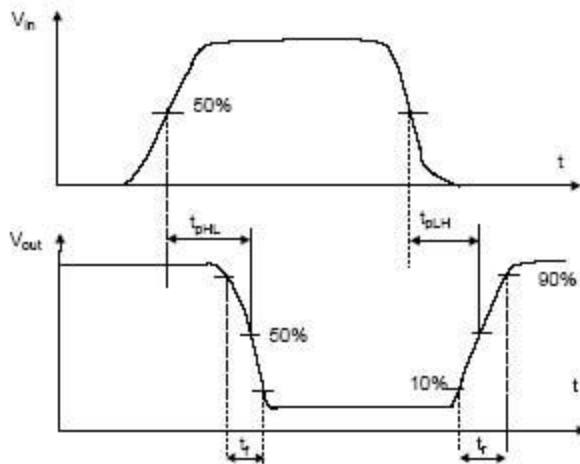
Due to the symmetry of the CMOS circuit the rise time can be similarly obtained as; For equally sized **n** and **p** transistors (where **βn=2βp**) **tf=tr**

Thus the fall time is faster than the rise time primarily due to different carrier mobilities associated with the p and n devices thus if we want **tf=tr** 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,

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

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



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

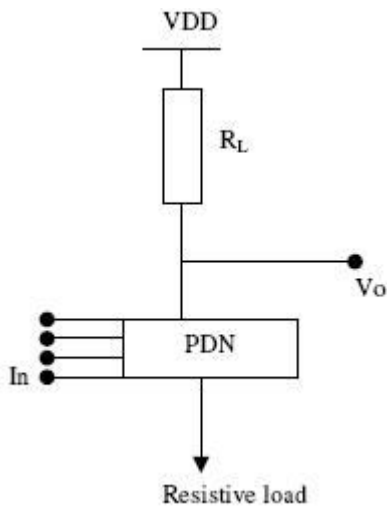
$$\tau_{PLH(actual)} = \sqrt{(\tau_{PLH})^2 + (t_r/2)^2}$$

$$\tau_{PFL(actual)} = \sqrt{(\tau_{PFL})^2 + (t_r/2)^2}$$

These are the rms values for the propagation delays.

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:

$$V_{OL} = \frac{R_D V_{DD}}{R_D + R_L}$$

where R_D = pulldown n/w resistance, R_L = load resistance.

To keep the low noise margin high it is important to choose $R_L \gg R_D$. 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 $R_L \geq 4R_D$. 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 $V_{DS}=V_{GS}$. Consequently $V_{DS} > V_{GS} - V_{tn}$.

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

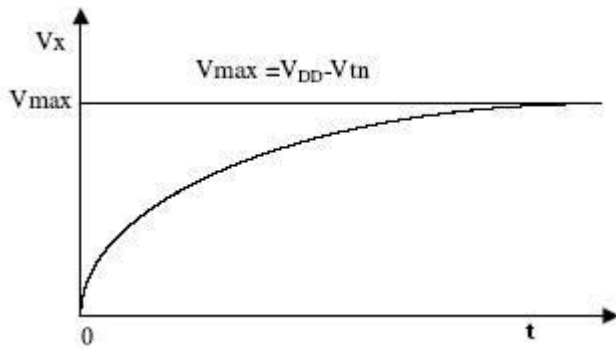
$$C_x (dV_x/dt) = (k_n/2)(V_{DD} - V_x - V_{tn})^2 \Rightarrow \int_0^t dt = (2C_x/k_n) \int_0^{V_x} dV_x / (V_{DD} - V_x - V_{tn})^2$$

$$\text{So, } t = (2C_x/k_n) [1/(V_{DD} - V_x - V_{tn}) - 1/(V_{DD} - V_{tn})]$$

The previous equation for $V_x(t)$ 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 $V_x(t)$ is plotted as a function of time in fig. The voltage rises from its initial value of 0 and reaches $V_{max} = V_{DD} - V_{tn}$ after a large time. The pass transistor will turn off when $V_x = V_{max}$. Since $V_{gs} = V_{tn}$. Therefore V_x can never attain V_{DD} during logic 1 transfer. Thus we can use buffering to overcome this problem.



Node Voltage V_x vs t

Logic “0” Transfer: Assume that the $V_x = I$

Initially, A logic “0” level is applied to the input terminal which corresponds to $V_{in} = I$. 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 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 $V_x(t)$ can be solved as –

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

Plot of $V_x(t)$ is shown in figure

Node Voltage V_x vs t

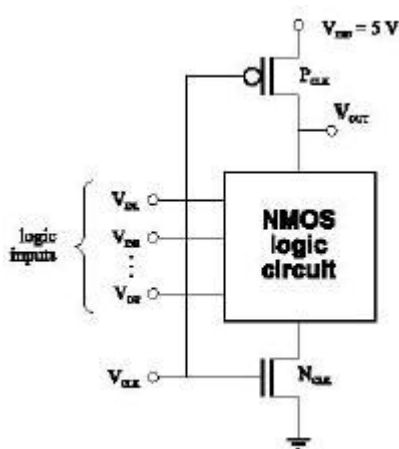
Dynamic Logic Circuits

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:

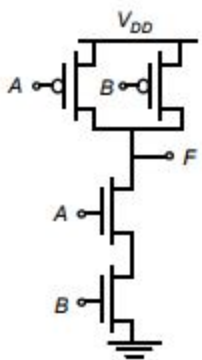
1. The penalty paid in such circuits is that the clock must run everywhere to each such block as shown in the diagram.
2. The major problem in such circuits is that the output node is at V_{dd} till the end of the precharge mode. Now if the **CLK** in the next block arrives earlier compared to the **CLK** 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 see, 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

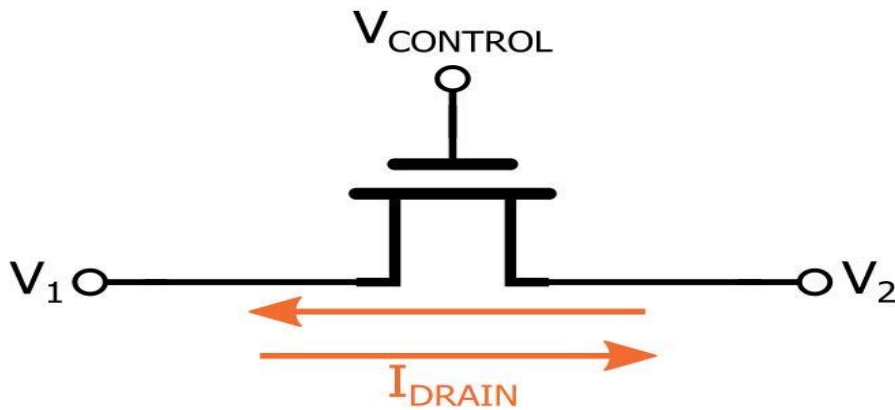
Two I/P NAND gate in complementary Static CMOS Style

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.

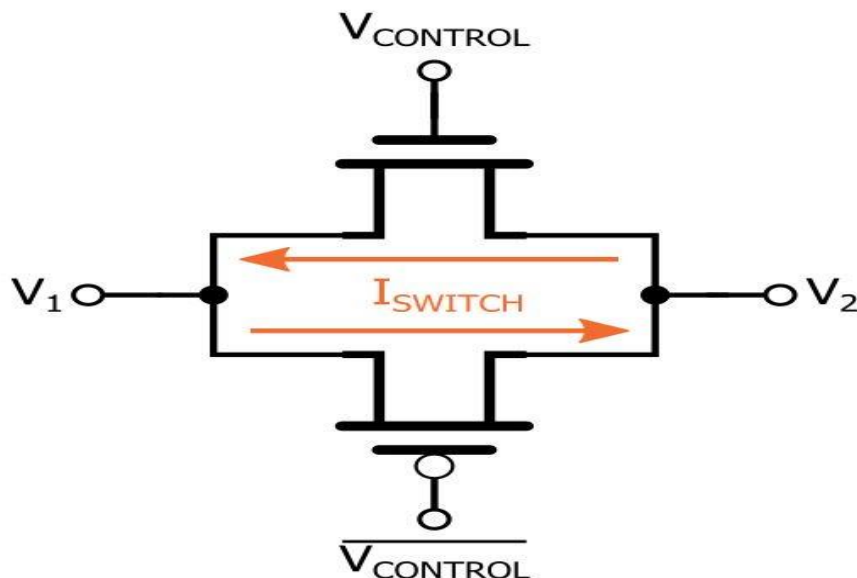
A single NMOS (or PMOS) transistor can be used as a voltage-controlled switch. The “circuit” (really just a single transistor) is the following:



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.

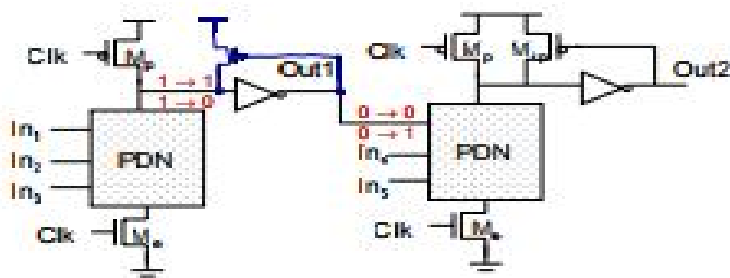
When you analyze and ponder this switch, you recognize a certain asymmetry. For example, if we are using this switch for pass-transistor logic, the NMOS can effectively pass a logic-low signal but not a full logic-high signal. Is it possible to modify the circuit in a way that will redress this asymmetry? 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.

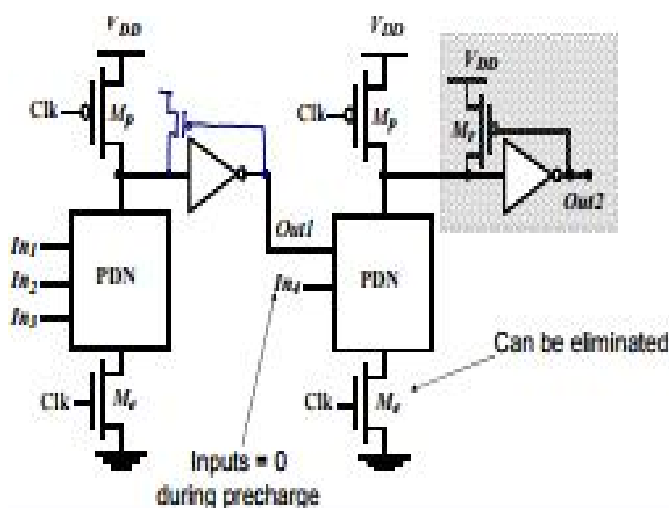
DOMINO LOGIC:



Properties of Domino Logic

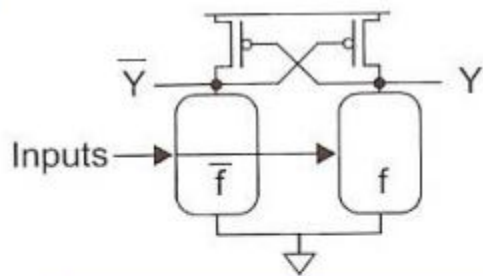
- Only non-inverting logic can be implemented %
- Very high speed f
- static inverter can be skewed, only L-H transition critical f
- Input capacitance reduced – smaller logical effort

DESIGNING WITH DOMINO LOGIC:

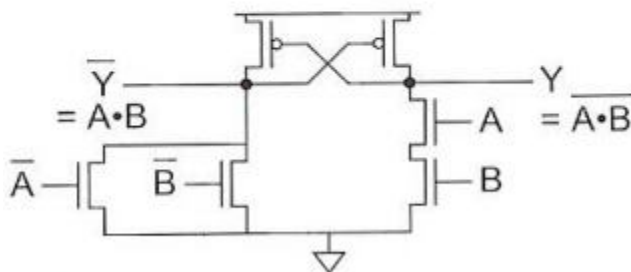


DIFFERENTIAL CASCODE VOLTAGE SWITCH LOGIC:

- Performance advantage of ratioed circuits without the extra power
- Requires complementary inputs – produces complementary outputs
- Operation – two nMOS arrays one for f , one for \bar{f} – cross-coupled load pMOS – one path is always active
 - since either f or \bar{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

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

<i>Before Scaling</i>	<i>After Constant Field Scaling</i>	<i>After Constant Voltage Scaling</i>
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}	$I'_{DS} = I_{DS}/s$	$I'_{DS} = I_{DS} * s$
PD	$P'_D = P_D/s^2$	$P'_D = P_D * s$

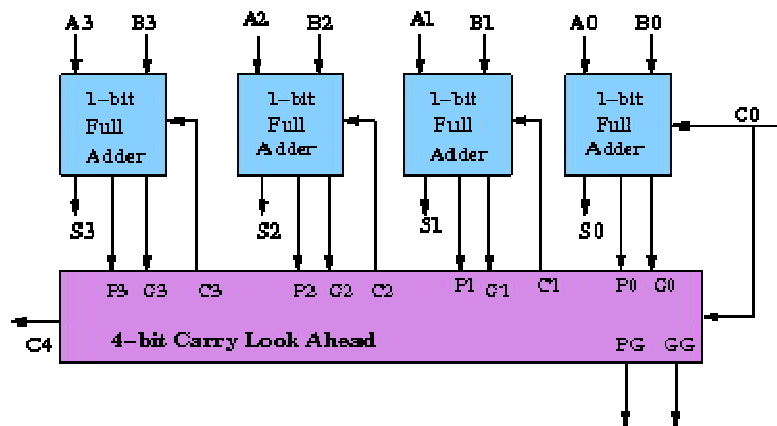
Where s = scaling parameter of MOS

UNIT III

VLSI IMPLEMENTATION STRATEGIES

Design of Carry Lookahead Adders :

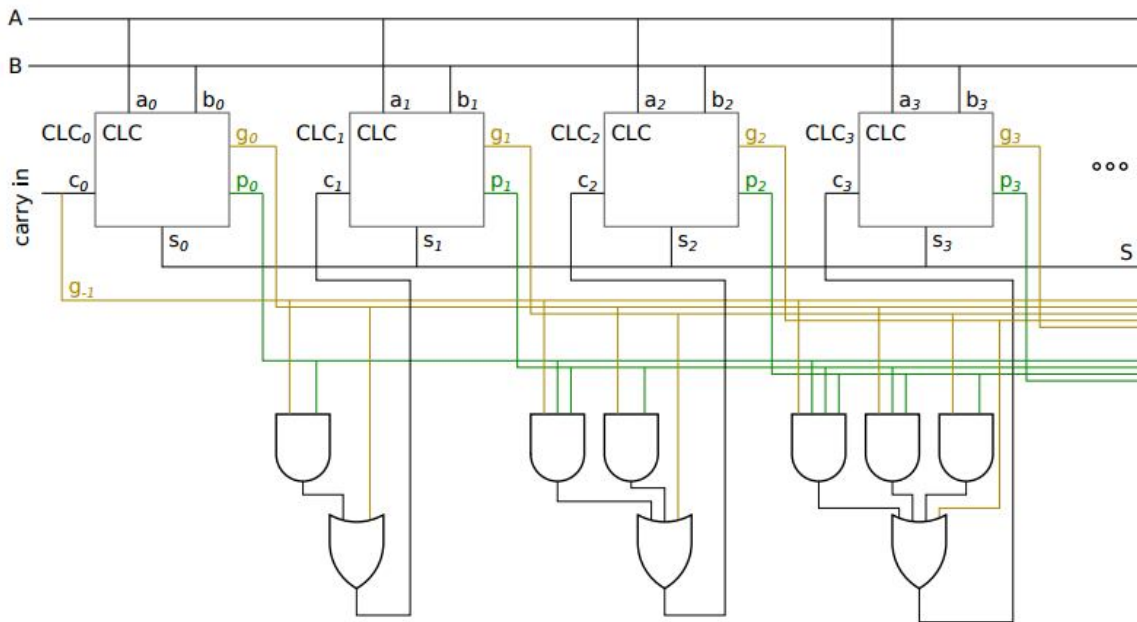
To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be Carry Propagator and Carry Generator. The **carry propagator** is propagated to the next level whereas the **carry generator** is used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



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

CARRY GENERATION LOGIC:

Types of carry generation logic (CGL): lookahead and ripple. With lookahead CGL adder above is a CLA. With ripple CGL adder above is equivalent to a ripple adder.



Boolean expressions for carry signals c_0 to c_3 :

$$c_0 = g_{-1}.$$

$$c_1 = g_{-1}p_0 + g_0.$$

$$c_2 = g_{-1}p_0p_1 + g_0p_1 + g_1.$$

$$c_3 = g_{-1}p_0p_1p_2 + g_0p_1p_2 + g_1p_2 + g_2.$$

Generalization of Lookahead Carry Generation

$$c_0 = g_{-1}.$$

$$c_1 = g_{-1}p_0 + g_0.$$

$$c_2 = g_{-1}p_0p_1 + g_0p_1 + g_1.$$

$$c_3 = g_{-1}p_0p_1p_2 + g_0p_1p_2 + g_1p_2 + g_2.$$

Generalizing we get

$$c_i = g_{-1}p_0p_1 \cdots p_{i-1} + g_0p_1p_2 \cdots p_{i-1} + g_1p_2p_3 \cdots p_{i-1} + \cdots + g_{i-2}p_{i-1} + g_{i-1}$$

$$= \sum_{j=-1}^{i-1} g_j \prod_{k=j+1}^{i-1} p_k$$

CARRY SAVE ADDERS:

The idea of delaying carry resolution until the end, or saving carries, is due to [John von Neumann](#).^[3]

Here is an example of a binary sum:

```
1011101010101101111000000001101
+ 1101111010101101101111101110111
```

Carry-save arithmetic works by abandoning the binary notation while still working to base 2. It computes the sum digit by digit, as

```
1011101010101101111000000001101
+ 1101111010101101101111101110111
= 21122120202022022122111011102212
```

The notation is unconventional but the result is still unambiguous. Moreover, given n adders (here, $n=32$ full adders), the result can be calculated after propagating the inputs through a single adder, since each digit result does not depend on any of the others.

If the adder is required to add two numbers and produce a result, carry-save addition is useless, since the result still has to be converted back into binary and this still means that carries have to propagate from right to left. But in large-integer arithmetic, addition is a very rare operation, and adders are mostly used to accumulate partial sums in a multiplication.

CARRY SAVE ACCUMULATORS:

The key to success is that at the moment of each partial addition we add three bits:

- 0 or 1, from the number we are adding.
- 0 if the digit in our store is 0 or 2, or 1 if it is 1 or 3.
- 0 if the digit to its right is 0 or 1, or 1 if it is 2 or 3.

To put it another way, we are taking a carry digit from the position on our right, and passing a carry digit to the left, just as in conventional addition; but the carry digit we pass to the left is the result of the *previous* calculation and not the current one. In each clock cycle, carries only have to move one step along, and not n steps as in conventional addition.

Because signals don't have to move as far, the clock can tick much faster.

There is still a need to convert the result to binary at the end of a calculation, which effectively just means letting the carries travel all the way through the number just as in a conventional adder. But if we have done 512 additions in the process of performing a 512-bit multiplication, the cost of that final conversion is effectively split across those 512 additions, so each addition bears 1/512 of the cost of that final "conventional" addition.

At each stage of a carry-save addition,

1. We know the result of the addition at once.
2. We *still do not know* whether the result of the addition is larger or smaller than a given number (for instance, we do not know whether it is positive or negative).

This latter point is a drawback when using carry-save adders to implement modular multiplication (multiplication followed by division, keeping the remainder only).

The carry-save unit consists of n [full adders](#), each of which computes a single sum and carry bit based solely on the corresponding bits of the three input numbers. Given the three n - bit numbers **a**, **b**, and **c**, it produces a partial sum **ps** and a shift-carry **sc**:

$$\{\displaystyle ps_i = a_i \oplus b_i \oplus c_i\} \quad \{\displaystyle sc_i = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)\}$$

The entire sum can then be computed by:

1. [Shifting](#) the carry sequence **sc** left by one place.
2. Appending a 0 to the front ([most significant bit](#)) of the partial sum sequence **ps**.
3. Using a [ripple carry adder](#) to add these two together and produce the resulting $n + 1$ -bit value.

MULTIPLIERS:

The Serial-parallel Multiplier

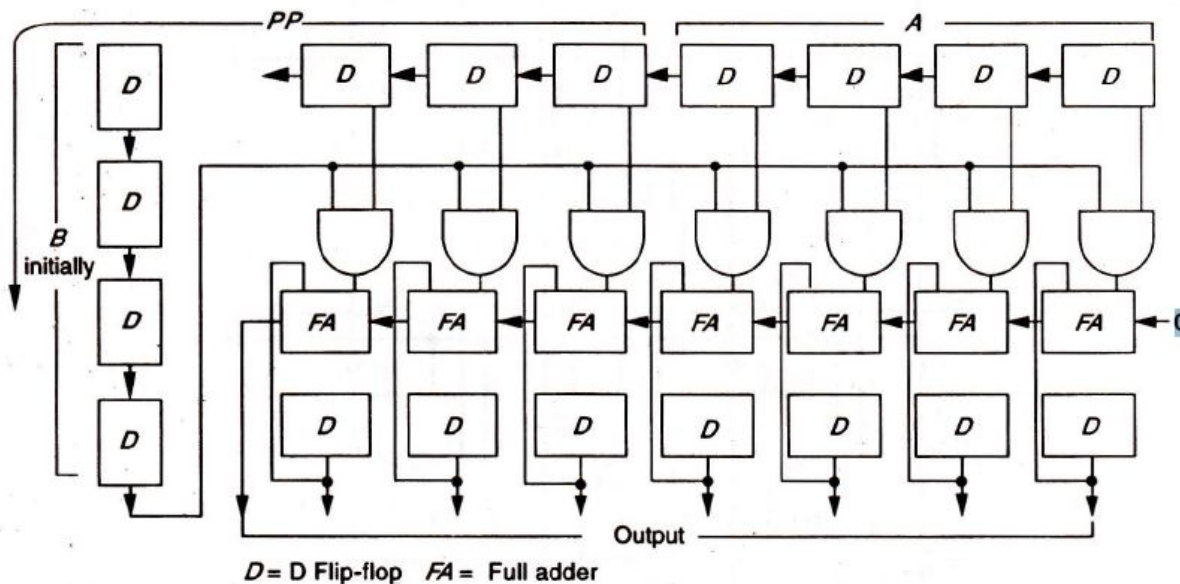
This multiplier is the simplest one, the multiplication being considered as a succession of additions.

$$\begin{aligned} \text{If} \quad A &= (a_n \ a_{n-1} \ a_{n-2} \ \dots \ a_0) \text{ and} \\ B &= (b_n \ b_{n-1} \ b_{n-2} \ \dots \ b_0) \end{aligned}$$

then the product $A.B$ may be expressed as

$$A.B = (A.2^n.b_n + A.2^{n-1}.b_{n-1} + A.2^{n-2}.b_{n-2} \dots A.2^0.b_0)$$

A possible form of this adder for multiplying four-bit quantities, based on this expression, is set out in the below Figure. Note that D indicates a D flip-flop simple and FA indicates a full adder--or adder bit slice. Number A is entered in the right-most 4-bits of the top row of D flip-flops which are connected to three further D flip-flops to form a 7-bit shift register to allow the multiplication of number A by $2^1, 2^2 \dots 2^n$, thus forming the *partial product* at each stage of the process.

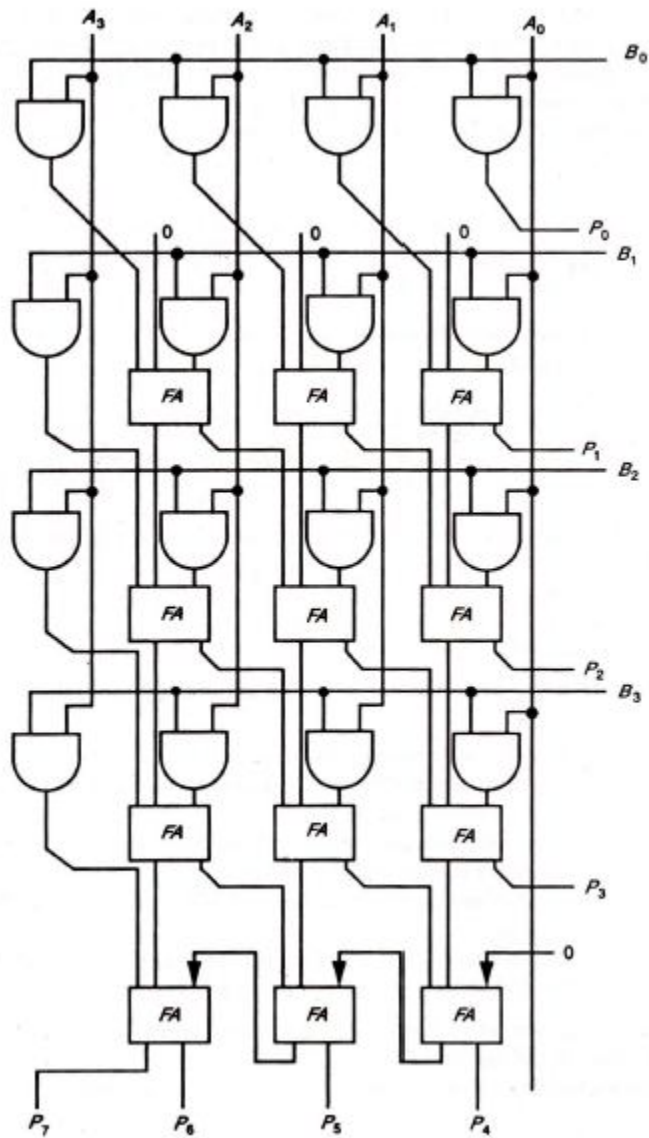


The structure under discussion here is suited only to positive or unsigned operands. If the operands are negative and two's complement encoded, then:

1. The most significant bit of B will have a negative weight and so a subtraction must be performed as the last step.
2. The most significant bit of A must be replicated since operand A must be expanded to $2N$ bits.

THE BRAUN ARRAY:

A relatively simple form of parallel adder is the Braun array. All partial products $A.b_k$ are computed in parallel, then collected through a cascaded array of carry save adders. At the bottom of the array, an adder is used to convert the carry save form to the required form of output. Completion time is fixed by the depth of the array, and by the carry propagation characteristics of the adder. Notice that this multiplier is suited only to positive operands. Negative operands can be handled, for example, by the Baugh-Wooley multiplier..



4 BIT Braun Multiplier.

Twos Complement Multipllcation Using the Baugh-Wooley Method

This technique has been developed to design multipliers that are regular in structure and suited for twos complement numbers.

Let us consider two numbers A and B:

$$A = (a_{n-1}, \dots, a_0) = -a_{n-1} \cdot 2^{n-1} + \sum_0^{n-2} a_i \cdot 2^i$$

$$B = (b_{n-1}, \dots, b_0) = -b_{n-1} \cdot 2^{n-1} + \sum_0^{n-2} b_i \cdot 2^i$$

The product $A.B$ is given by:

$$A.B = a_{n-1} \cdot b_{n-1} \cdot 2^{n-2} + \sum_0^{n-2} \sum_0^{n-2} a_i \cdot b_j \cdot 2^{i+j} - a_{n-1} \sum_0^{n-2} b_i \cdot 2^{n+i-1} - b_{n-1} \sum_0^{n-2} a_i \cdot 2^{n+i-1}$$

If we use this form, it may be seen that subtraction operations are needed as well as addition. However, the negative terms may be rewritten, for example:

$$a_{n-1} \sum_0^{n-2} b_i \cdot 2^{n+i-1} = a_{n-1} \cdot \left(-2^{n-2} + 2^{n-1} + \sum_0^{n-2} \bar{b}_i \cdot 2^{n+i-1} \right)$$

Using this approach, $A.B$ becomes

$$\begin{aligned} A.B = & a_{n-1} \cdot b_{n-1} \cdot 2^{n-2} + \sum_0^{n-2} \sum_0^{n-2} a_i \cdot b_j \cdot 2^{i+j} + b_{n-1} \left(-2^{n-2} + 2^{n-1} + \sum_0^{n-2} \bar{a}_i \cdot 2^{n+i-1} \right) \\ & + a_{n-1} \left(-2^{n-2} + 2^{n-1} + \sum_0^{n-2} \bar{b}_i \cdot 2^{n+i-1} \right) \end{aligned}$$

This equation may be put in a more convenient form by recognizing that

$$-(b^{n-1} + a^{n-1}) \cdot 2^{2n-2} = -2^{2n-1} + (\bar{a}_{n-1} + \bar{b}_{n-1}) \cdot 2^{2n-2}$$

Thus, AB is given by

$$\begin{aligned} A.B = & 2^{2n-1} + (\bar{a}_{n-1} + \bar{b}_{n-1} + a^{n-1} \cdot b^{n-1}) \cdot 2^{2n-2} \\ & + \sum_0^{n-2} \sum_0^{n-2} a_i \cdot b_j \cdot 2^{i+j} + (a_{n-1} + b_{n-1}) \cdot 2^{n-1} \\ & + \sum_0^{n-2} b_{n-1} \cdot \bar{a}_i \cdot 2^{n+1-j} + \sum_0^{n-2} a_{n-1} \cdot \bar{b}_i \cdot 2^{n+1-i} \end{aligned}$$

Since A and B are n -bit operands, their product may extend to $2n$ -bits. The first, most significant, bit is taken into account by the first term -2^{2n-1} which is fed to the multiplier as a 1 in the most significant cell. In serial-parallel multipliers there are as many idle clock cycles as there are 0s in the multiplicand and the same situation applies in Braun and Baugh-Wooley arrays. For this reason, it may be useful to introduce pipelining concepts between successive lines of the

array. The clock speed of the pipeline is limited by the speed of the output adder, but it is possible to introduce further pipelining between the adder cells giving rise to the systolic array multiplier.

FPGA: A Field-Programmable Gate Array (FPGA) is a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions.

ASIC: An application-specific integrated circuit (ASIC) is an integrated circuit designed for a

particular use, rather than intended for general-purpose use. Processors, RAM, ROM, etc are examples of ASICs.

FPGA vs ASIC

Speed

ASIC rules out FPGA in terms of speed. As ASIC are designed for a specific application they can be optimized to maximum, hence we can have high speed in ASIC designs. ASIC can have high speed clocks.

Cost

FPGAs are cost effective for small applications. But when it comes to complex and large volume designs (like 32-bit processors) ASIC products are cheaper.

Size/Area

FPGA are contains lots of LUTs, and routing channels which are connected via bit streams(program). As they are made for general purpose and because of re-usability. They are in-general larger designs than corresponding ASIC design. For example, LUT gives you both registered and non-register output, but if we require only non-registered output, then its a waste of having a extra circuitry. In this way ASIC will be smaller in size.

Power

FPGA designs consume more power than ASIC designs. As explained above the unwanted circuitry results wastage of power. FPGA wont allow us to have better power optimization. When it comes to ASIC designs we can optimize them to the fullest.

Time to Market

FPGA designs will till less time, as the design cycle is small when compared to that of ASIC designs. No need of layouts, masks or other back-end processes. Its very simple: Specifications -- HDL + simulations -- Synthesis -- Place and Route (along with static-analysis) -- Dump code onto FPGA and Verify. When it comes to ASIC we have to do floor planning and also advanced verification. The FPGA design flow eliminates the complex and time-consuming floor planning, place and route, timing analysis, and mask / re-spin stages of the project since the design logic is already synthesized to be placed onto an already verified, characterized FPGA device.

Type of Design

ASIC can have mixed-signal designs, or only analog designs. But it is not possible to design them using FPGA chips.

Customization

ASIC has the upper hand when comes to the customization. The device can be fully customized as ASICs will be designed according to a given specification. Just imagine implementing a 32-bit processor on a FPGA!

Prototyping

Because of re-usability of FPGAs, they are used as ASIC prototypes. ASIC design HDL code is first dumped onto a FPGA and tested for accurate results. Once the design is error free then it is taken for further steps. Its clear that FPGA may be needed for designing an ASIC.

Non Recurring Engineering/Expenses

NRE refers to the one-time cost of researching, designing, and testing a new product, which is generally associated with ASICs. No such thing is associated with FPGA. Hence FPGA designs are cost effective.

Simpler Design Cycle

Due to software that handles much of the routing, placement, and timing, FPGA designs have smaller designed cycle than ASICs.

More Predictable Project Cycle

Due to elimination of potential re-spins, wafer capacities, etc. FPGA designs have better project cycle.

Tools

Tools which are used for FPGA designs are relatively cheaper than ASIC designs.

Re-Usability

A single FPGA can be used for various applications, by simply reprogramming it (dumping new HDL code). By definition ASIC are application specific cannot be reused.

Field-Programmable Gate Array

In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

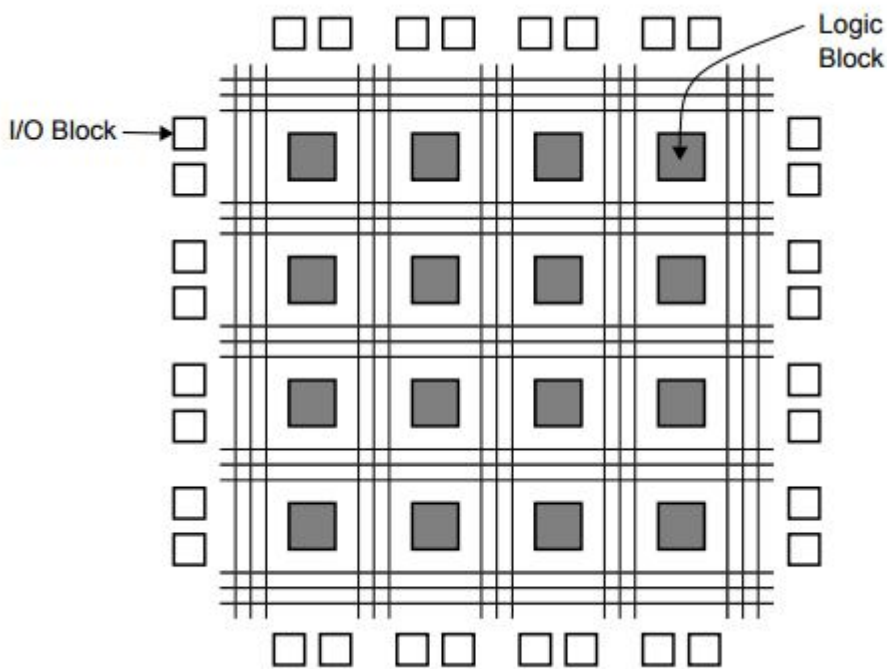
Applications

ASIC prototyping: Due to high cost of ASIC chips, the logic of the application is first verified by dumping HDL code in a FPGA. This helps for faster and cheaper testing. Once the logic is verified then they are made into ASICs.

- Very useful in applications that can make use of the massive parallelism offered by their architecture. Example: code breaking, in particular brute-force attack, of cryptographic algorithms.
- FPGAs are used for computational kernels such as FFT or Convolution instead of a microprocessor.
- Applications include digital signal processing, software-defined radio, aerospace and defense systems, medical imaging, computer vision, speech recognition, cryptography, bio-informatics, computer hardware emulation and a growing range of other areas.

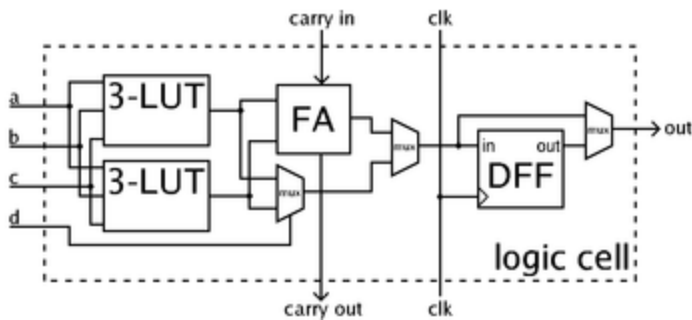
Architecture

FPGA consists of large number of "configurable logic blocks" (CLBs) and routing channels. Multiple I/O pads may fit into the height of one row or the width of one column in the array. In general all the routing channels have the same width. The block diagram of FPGA architecture is shown below.



CLB: The CLB consists of an n-bit look-up table (LUT), a flip-flop and a 2x1 mux. The value n is manufacturer specific. Increase in n value can increase the performance of a FPGA. Typically n is 4. An n-bit lookup table can be implemented with a multiplexer whose select lines are the inputs of the LUT and whose inputs are constants. An n-bit LUT can encode any n-input Boolean function

by modeling such functions as truth tables. This is an efficient way of encoding Boolean logic functions, and LUTs with 4-6 bits of input are in fact the key component of modern FPGAs. The block diagram of a CLB is shown below.



Each CLB has n-inputs and only one output, which can be either the registered or the unregistered LUT output. The output is selected using a 2x1 mux. The LUT output is registered using the flip-flop (generally D flip-flop). The clock is given to the flip-flop, using which the output is registered. In general, high fanout signals like clock signals are routed via special-purpose dedicated routing networks, they and other signals are managed separately.

Routing channels are programmed to connect various CLBs. The connecting done according to the design. The CLBs are connected in such a way that logic of the design is achieved.

FPGA Programming

The design is first coded in HDL (Verilog or VHDL), once the code is validated (simulated and synthesized). During synthesis, typically done using tools like Xilinx ISE, FPGA Advantage, etc, a technology-mapped net list is generated. The net list can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated is used to (re)configure the FPGA. Once the FPGA is (re)configured, it is tested. If there are any issues or modifications, the original HDL code will be modified and then entire process is repeated, and FPGA is reconfigured.

FULL CUSTOM DESIGN:

- A Full custom design is one which includes some (possibly all) logic cells that are customized and all mask layers that are customized.

- A microprocessor is an example of a full-custom IC . Designers spend many hours squeezing the most out of every last square micron of microprocessor chip space by hand.
- Customizing all of the IC features in this way allows designers to include analog circuits, optimized memory cells, or mechanical structures on an IC, for example. Full-custom ICs are the most expensive to manufacture and to design.
- The manufacturing lead time (the time required just to make an IC not including design time) is typically eight weeks for a full-custom IC.
- These specialized full-custom ICs are often intended for a specific application so, we might call some of them as full-custom ASICs
- In a full-custom design One has to use full-custom design if the technology is new or so specialized that there are no existing cell libraries or because the technology is so specialized that some circuits must be custom designed.
- Fewer and fewer full-custom ICs are being designed because of the problems with these special parts of the technology.
- The growing member of this family, now a days is the mixed analog/digital design,
- an engineer designs some or all of the logic cells, circuits, or layout specifically for one application. This means the designer avoids using pretested and pre characterized cells for all or part of that design.
- This might be because existing cell libraries are not fast enough, or the logic cells are not small enough or consume too much power.

SEMI CUSTOM DESIGN:

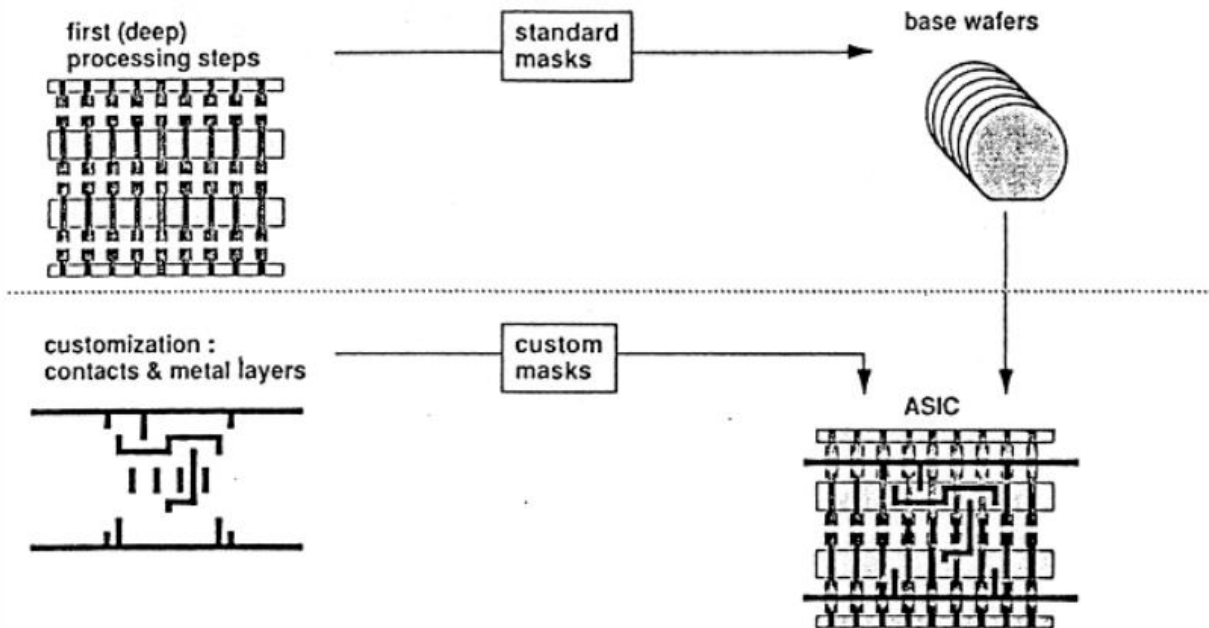
GATE ARRAY BASED DESIGN:

In view of the fast prototyping capability, the gate array (GA) comes after the FPGA.

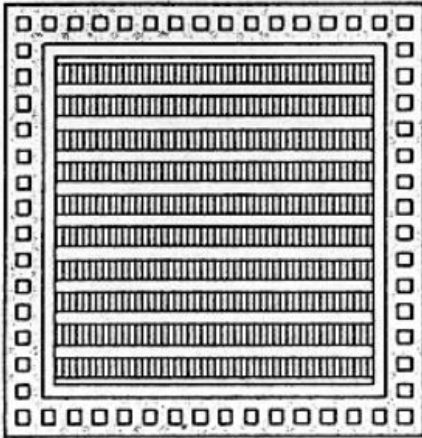
Design implementation of

- FPGA chip is done with user programming,
- Gate array is done with metal mask design and processing.
- Gate array implementation requires a two-step manufacturing process:
 - a) The first phase, which is based on generic (standard) masks, results in an array of uncommitted transistors on each GA chip.
 - b) These uncommitted chips can be customized later, which is completed by defining the metal interconnects between the transistors of the array.

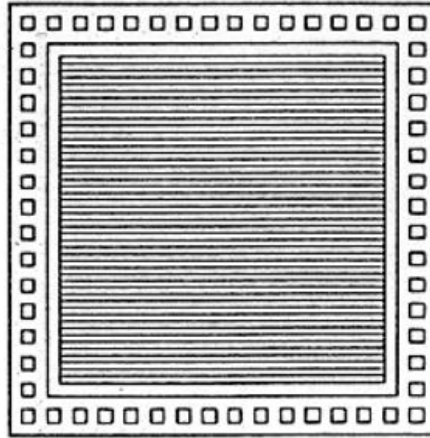
two-step manufacture :



CHANNELLED VS CHANNEL- LESS (SOG) APPROACHES:



routing problem is simpler
OK with only one metal



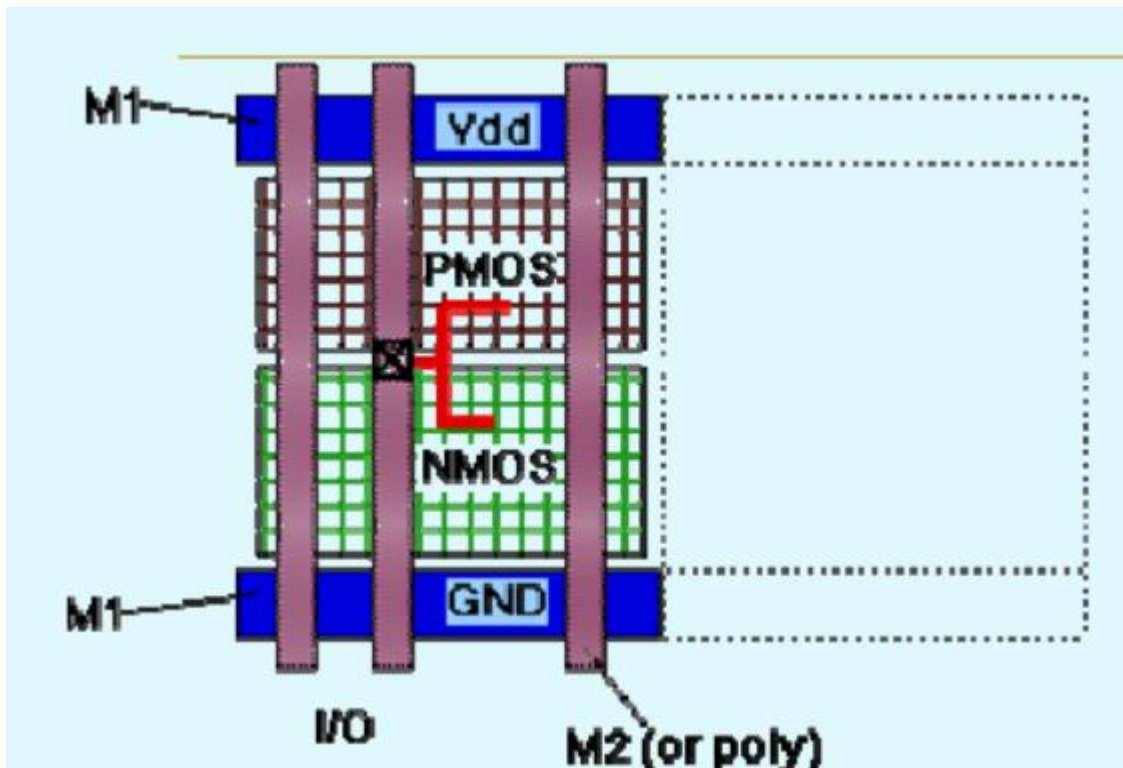
flexibility in channel definition (position & width)
over-the-cell routing
higher packing density
RAM-compatible
supports variable-height cells & macrocells
now universally used

STANDARD CELL BASED DESIGN:

Each cell is designed with a fixed height.

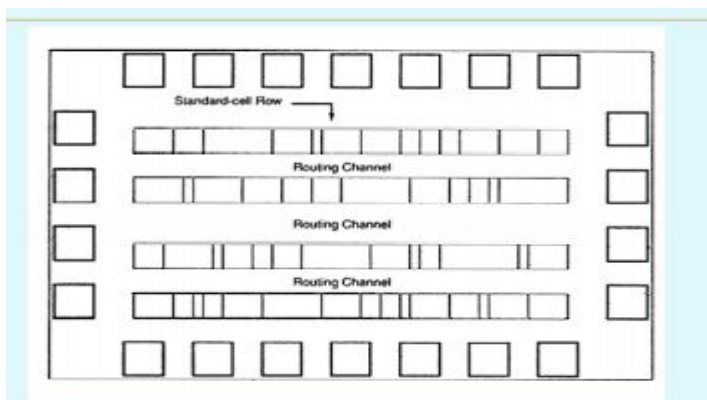
- To enable automated placement of the cells, and
- Routing of inter-cell connections.
- A number of cells can be abutted side-by-side to form rows.
- The power and ground rails typically run parallel to upper and lower boundaries of cell.
- Neighboring cells share a common power and ground bus.
- nMOS transistors are located closer to the ground rail while the pMOS transistors are placed closer to the power rail. The input and output pins are located on the upper and lower boundaries of the cell.

STANDARD CELLS:



Floorplan for Standard Cell Design

Inside the I/O frame which is reserved for I/O cells, the chip area contains rows or columns of standard cells. Between cell rows are channels for dedicated inter-cell routing. Over-the-cell routing is also possible. The physical design and layout of logic cells ensure that When placed into rows, their heights match. Neighboring cells can be abutted side-by-side, which provides natural connections for power and ground lines in each row.



UNIT IV

CMOS TESTING

Importance of testing:

Tests fall into three main categories. The first set of tests verifies that the chip performs its intended function. These tests, called *functionality tests* or *logic verification*, are run before tapeout to verify the functionality of the circuit. The second set of tests are run on the first batch of chips that return from fabrication. These tests confirm that the chip operates as it was intended and help debug any discrepancies. They can be much more extensive than the logic verification tests because the chip can be tested at full speed in a system. For example, a new microprocessor can be placed in a prototype motherboard to try to boot the operating system. This *silicon debug* requires creative detective work to locate the cause of failures because the designer has much less visibility into the fabricated chip compared to during design verification. The third set of tests verify that every transistor, gate, and storage element in the chip functions correctly. These tests are conducted on each manufactured chip before shipping to the customer to verify that the silicon is completely intact. These are called *manufacturing tests*. In some cases, the same tests can be used for all three steps, but often it is better to use one set of tests to chase down logic bugs and another, separate set optimized to catch manufacturing defects.

Testing a die (chip) can occur at the following levels:

Wafer level

Packaged chip level

Board level

System level

Field level

Obviously, if faults can be detected at the wafer level, the cost of manufacturing is lower. In an extreme example, Intel failed to correct a logic bug in the Pentium floating-point divider until more than 4 million units had shipped in 1994. IBM halted sales of Pentium-based computers and Intel

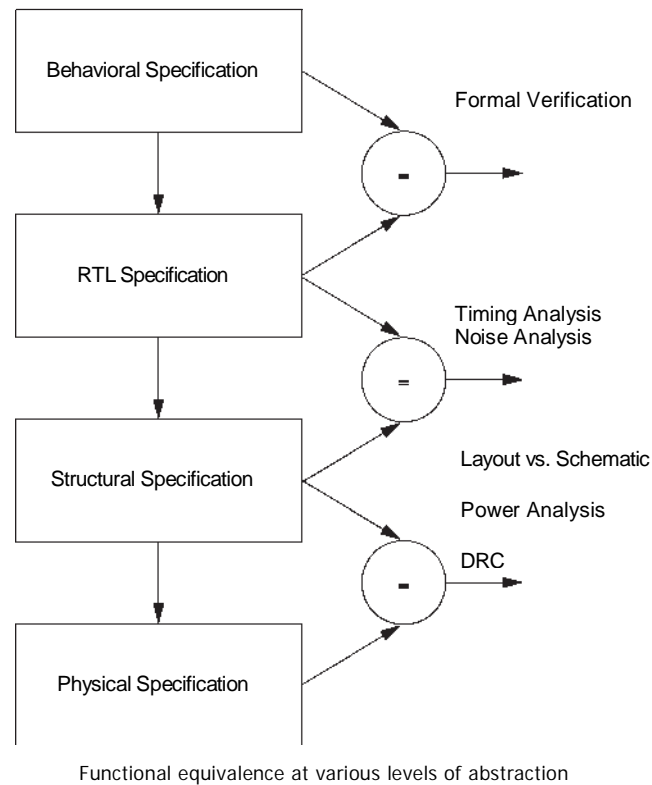
was forced to recall the flawed chips. The mistake and lack of prompt response cost the company an estimated \$450 million.

It is interesting to note that most first-time silicon result from problems functionality of the design; i.e., the exactly what the simulator said it but for some reason (almost always error) this functionality is not what the system expects.

Logic Verification

verification tests were required to synthesized gate description was equivalent to the source RTL. Figure

that we may want to prove that the RTL is equivalent to the design specification at a higher behavioral or specification level of abstraction. The behavioral specification might be a verbal description, a plain language textual specification, a description in some high-level computer language such as C, a program in a system-modeling language such as SystemC, or a hardware description language such as VHDL or Verilog, or simply a table of inputs and required outputs. Often, designers produce a *golden model* in one of the previously mentioned formats and it becomes the reference against which all other representations are checked. Functional equivalence involves running a simulator on the two descriptions of the chip (e.g., one at the gate level and one at a functional level) and ensuring that the outputs are equivalent at some convenient check points in time for all inputs applied. This is most conveniently done in an HDL by employing a *test bench*; i.e., a wrapper that surrounds a module and provides for stimulus and automated checking. The most detailed check might be on a cycle-by-cycle basis. Increasingly, verification involves real-time or near real-time emulation in an FPGA-based system to confirm system-level performance *in situ*; i.e., in the actual system that will use the end chip. This is recommended because of the increasing level of complexity of chips and the systems they implement.



failures of with the chip does would do, human the rest of

prove that a functionally 15.1 shows

You can check functional equivalence through simulation at various levels of the design hierarchy. If the description is at the RTL level, the behavior at a system level may be able to be fully verified. For instance, in the case of a microprocessor, you can boot the operating system and run key programs for the behavioral description. However, this might be impractical (due to long

simulation times) for a gate-level model and even harder for a transistor-level model. The way out of this impasse is to use the hierarchy inherent within a system to verify chips and modules within chips. That, combined with well-defined modular interfaces, goes a long way in increasing the likelihood that a system composed of many VLSI chips will be first-time functional.

The best advice with respect to writing functional tests is to simulate as closely as possible the way in which the chip or system will be used in the real world. Often, this is impractical due to slow simulation times and extremely long verification sequences. One approach is to move up the simulation hierarchy as modules become verified at lower levels. For instance, you could replace the gate-level adder and register modules in a video filter with functional models and then in turn replace the filter itself with a functional model. At each level, you can write small tests to verify the equivalence between the new higher-level functional model and the lower-level gate or functional level. At the top level, you can surround the filter functional model with a software environment that models the real-world use of the filter. Verification at the top chip level using an FPGA emulator offers several advantages over simulation and, for that matter, the final chip implementation. Most noticeably, the emulation speed can be near if not real time. This means that the actual analog signals (if used) can be interfaced with the chip. Additionally, to assess system performance, you can introduce fine levels of observation and monitoring that might not be included in the final chip. For instance, you could include a bit-error rate circuit in a communication modem to aid performance optimization.

Manufacturing Tests

Whereas verification or functionality tests seek to confirm the function of a chip as a whole, manufacturing tests are used to verify that every gate operates as expected. The need to do this arises from a number of manufacturing defects that might occur during either chip fabrication or accelerated life testing (where the chip is stressed by over-voltage and over-temperature operation). Typical defects include the following:

- Layer-to-layer shorts (e.g., metal-to-metal)

- Discontinuous wires (e.g., metal thins when crossing vertical topology jumps)

- Missing or damaged vias

- Shorts through the thin gate oxide to the substrate or well

These in turn lead to particular circuit maladies, including the following:

- Nodes shorted to power or ground
- Nodes shorted to each other

Inputs floating/outputs disconnected

Tests are required to verify that each gate and register is operational and has not been compromised by a manufacturing defect. Tests can be carried out at the wafer level to cull out bad dies, or can be left until the parts are packaged. This decision would normally be determined by the yield and package cost. If the yield is high and the package cost low (i.e., a plastic package), then the part can be tested only once after packaging. However, if the wafer yield was lower and the package cost high (i.e., an expensive ceramic package), it is more economical to first screen bad dice at the wafer level. The length of the tests at the wafer level can be shortened to reduce test time based on experience with the test sequence.

Apart from the verification of internal gates, I/O integrity is also tested, with the following tests being completed:

I/O levels (i.e., checking noise margin for TTL, ECL, or CMOS I/O pads)

Speed test

With the use of on-chip test structures described in Section 15.6, full-speed wafer testing can be completed with a minimum of connected pins. This can be important in reducing the cost of the wafer test fixture.

In general, manufacturing test generation assumes the function of the circuit/chip is correct. It requires ways of exercising all gate inputs and monitoring all gate outputs.

FUNCTIONALITY TESTS

Functionality tests verify that the chip performs its intended function. These tests assert that all the gates in the chip, acting in concert, achieve a desired function. These tests are usually used early in the design cycle to verify the functionality of the circuit.

OCCURANCE OF DIFFERENT FAULTS:

Examples of *physical defects* include:

Defects in silicon substrate

Photolithographic defects

Mask contamination and scratches

Process variations and abnormalities

Oxide defects

The physical defects can cause electrical faults and logical faults.

The *electrical* faults include:

Shorts (bridging faults)

Opens

Transistor stuck-on, stuck-open

Resistive shorts and opens

Excessive change in threshold voltage

Excessive steady-state currents

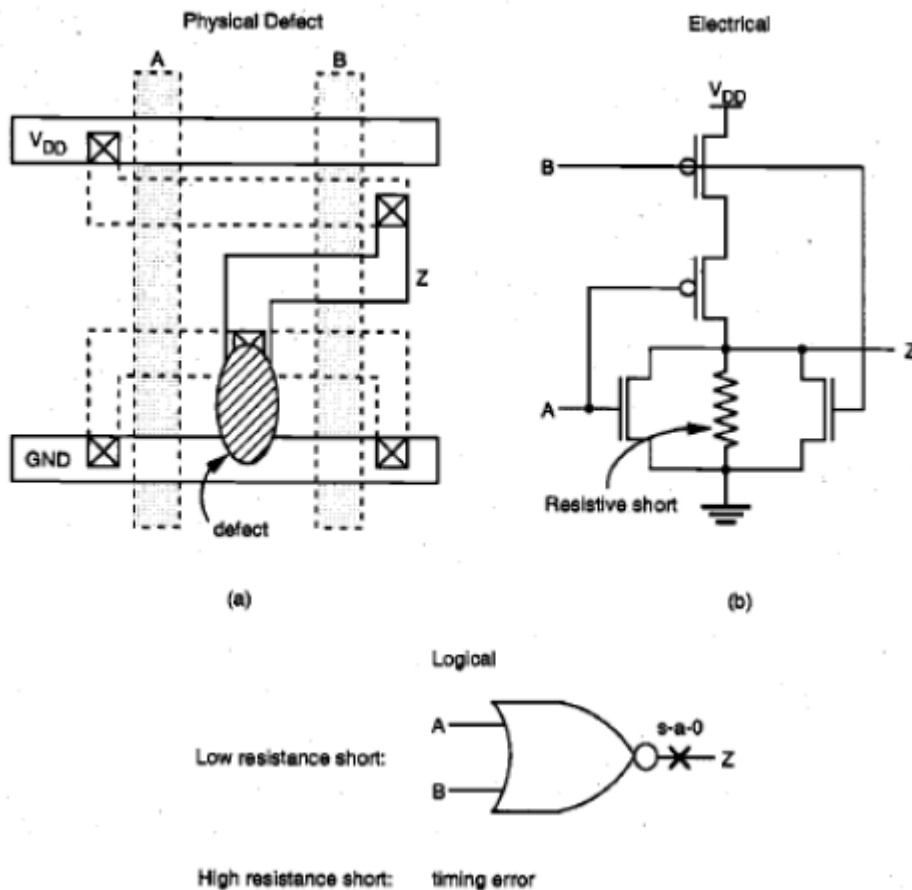
The electrical faults in turn can be translated into logical faults.

The *logical* faults include:

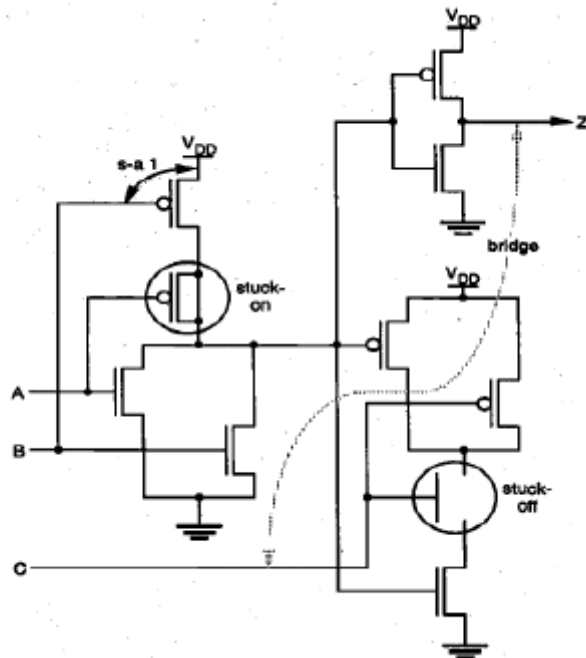
Logical stuck-at-0 or stuck-at-1

Slower transition (delay fault)

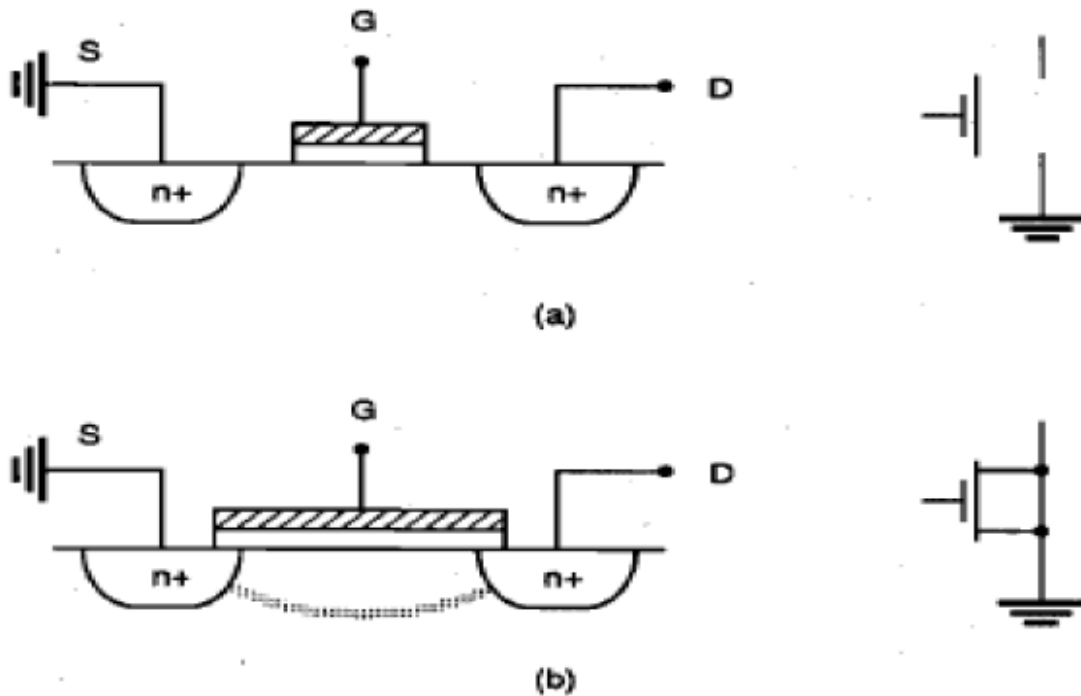
AND-bridging, OR-bridging



NAND2, and inverter gates. In this circuit, the input line B can be stuck-at- 1 (s-a- 1), since some part of the input line is shorted to the power rail. The pMOS transistor of the first stage NOR2 gate is stuck-on due to a process problem that causes a short between its source and drain terminals. The top nMOS transistor in the NAND2 gate, on the other hand, is stuck-open due to either an incomplete contact (open) of the source or drain node or due to a large separation of drain or source diffusion from the gate, which causes permanent turn-off of the transistor regardless of the input C value. The stuck-on and stuck-open faults are elaborated on in Fig. The bridging fault between the output line of the inverter and the input line C can be due to a fabrication defect which causes a short between any two parts of the two lines. Although in the circuit diagram, these two lines are seemingly far apart, in the actual layout, some parts of these two lines can be close to each other. In such a layout, these two lines can be shorted due to underetching in the line patterning process.



- Improper estimation of on –chip interconnect delays and other timing considerations.
- Excessive variations in the fabrication process which cause significant variations in circuit delays and clock skews.
- Opens in metal lines connecting parallel transistors which make the effective transistor size much smaller.
- Ageing effects such as hot –carrier induced delay increase.



The task of detecting delay faults is even more subtle than detecting functional faults in steady state. The functional test is usually done at speeds lower than the target speed due to the limitations of the testers. Special clocking is used to apply delay tests on a slow tester. The fault models mentioned above are used in fault simulation aimed at

- Test generation,
- Construction of fault dictionaries, or
- Circuit analysis in the presence of faults.

Each fault dictionary stores the expected output response of every faulty circuit to a particular test vector corresponding to a particular simulated fault.

CONTROLLABILITY AND OBSERVABILITY:

Controllability : The ability to set nets, nodes, gate inputs or outputs or sequential elements to a known logic state.

Observability : The ability to observe nets, nodes, gate inputs or outputs or sequential elements to a known logic state.

Controllability (C1) : Controllability C1 is the probability of a signal value on line l being set to 1 by a random vector . Controllability (C0) : Controllability C0 is the probability of a signal value on line l being set to 0 by a random vector .

SCOAP (Sandia Controllability / Observability analysis program)

Combinational 0-controllability, $CC0(n)$

Combinational 1-controllability, $CC1(n)$

Combinational observability, $C0(n)$

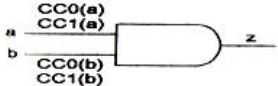
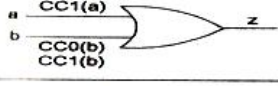
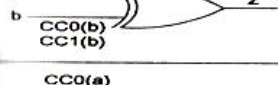
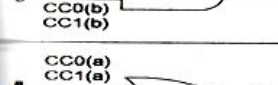
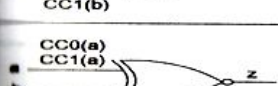
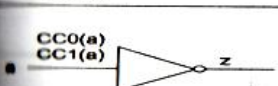
Sequential 0-controllability, $SC0(n)$

Sequential 1-controllability, $SC1(n)$

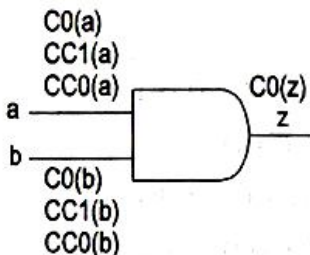

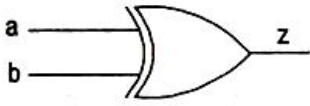
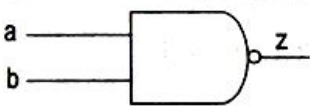


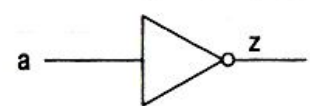
Sequential observability, $S0(n)$

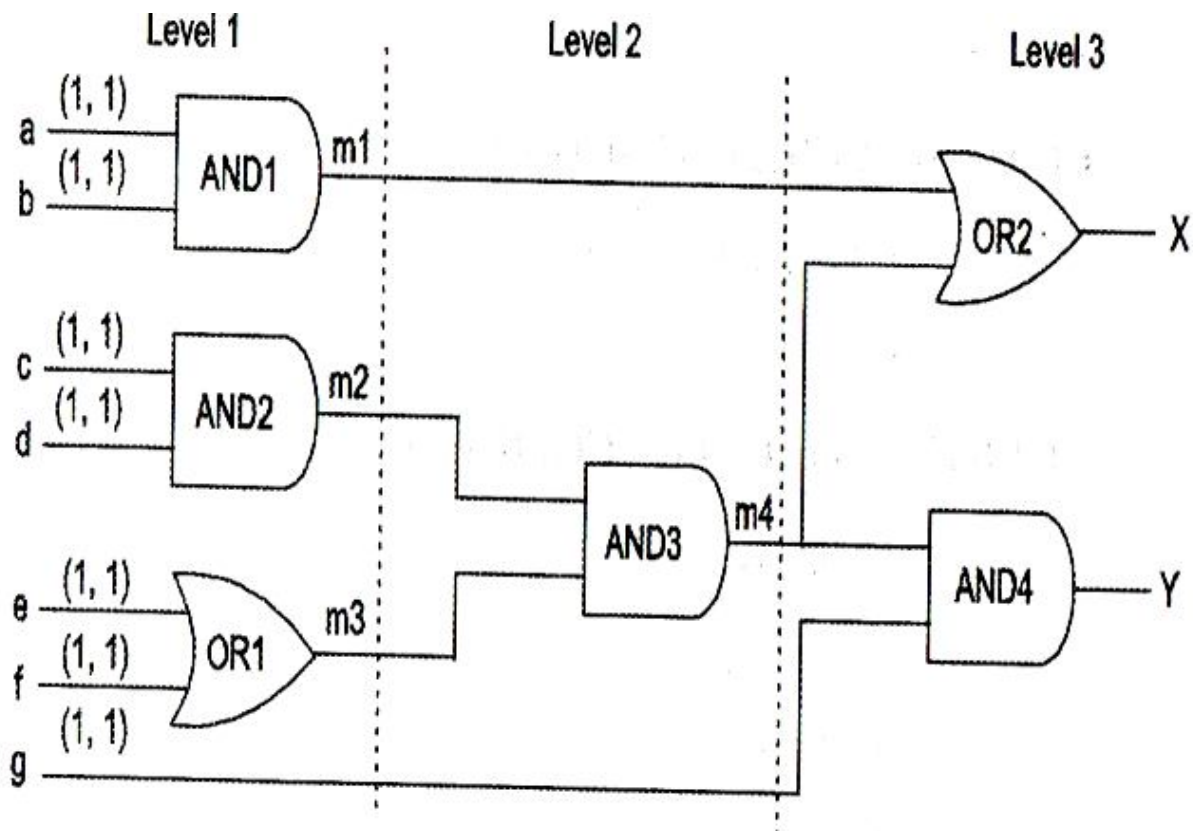
If a logic gate output is produced by setting only one input to a controlling value then Output controllability = $\min(\text{input controllabilities}) + 1$ If a logic gate output can only be produced by setting all inputs to a non-controlling value then

Output controllability = $\sum(\text{input controllabilities}) + 1$

Logic Gate	Truth Table	SCOAP Controllability Calculation															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	z	0	0	0	0	1	0	1	0	0	1	1	1	$CC0(z) = \min(CC0(a), CC0(b)) + 1$ $CC1(z) = CC1(a) + CC1(b) + 1$
a	b	z															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	z	0	0	0	0	1	1	1	0	1	1	1	1	$CC0(z) = CC0(a) + CC0(b) + 1$ $CC1(z) = \min(CC1(a), CC1(b)) + 1$
a	b	z															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	z	0	0	0	0	1	1	1	0	1	1	1	0	$CC0(z) = \min(CC0(a) + CC0(b),$ $CC1(a) + CC1(b)) + 1$ $CC1(z) = \min(CC0(a) + CC1(b),$ $CC1(a) + CC0(b)) + 1$
a	b	z															
0	0	0															
0	1	1															
1	0	1															
1	1	0															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	z	0	0	1	0	1	1	1	0	1	1	1	0	$CC0(z) = CC1(a) + CC1(b) + 1$ $CC1(z) = \min(CC0(a), CC0(b)) + 1$
a	b	z															
0	0	1															
0	1	1															
1	0	1															
1	1	0															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	z	0	0	1	0	1	0	1	0	0	1	1	0	$CC0(z) = \min(CC1(a), CC1(b)) + 1$ $CC1(z) = CC0(a) + CC0(b) + 1$
a	b	z															
0	0	1															
0	1	0															
1	0	0															
1	1	0															
	<table><tr><td>a</td><td>b</td><td>z</td></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	z	0	0	1	0	1	0	1	0	0	1	1	1	$CC0(z) = \min(CC0(a) + CC1(b),$ $CC1(a) + CC0(b)) + 1$ $CC1(z) = \min(CC0(a) + CC0(b),$ $CC1(a) + CC1(b)) + 1$
a	b	z															
0	0	1															
0	1	0															
1	0	0															
1	1	1															
	<table><tr><td>a</td><td>z</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	a	z	0	1	1	0	$CC0(z) = CC1(a) + 1$ $CC1(z) = CC0(a) + 1$									
a	z																
0	1																
1	0																

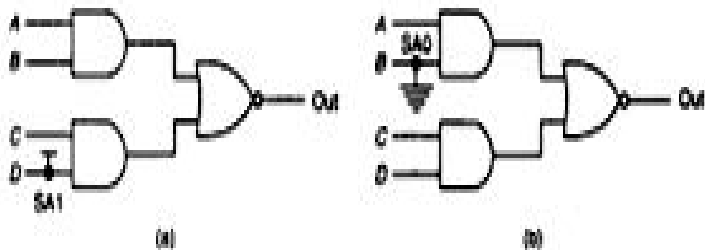
Observability Observability = observability of the output + non controlling value + 1

Logic Gates	Observability Calculation
	$C0(a) = C0(z) + CC1(b) + 1$ $C0(b) = C0(z) + CC1(a) + 1$
	$C0(a) = C0(z) + CC0(b) + 1$ $C0(b) = C0(z) + CC0(a) + 1$
	$C0(a) = C0(z) + \min(CC0(b), CC1(b)) + 1$ $C0(b) = C0(z) + \min(CC0(a), CC1(a)) + 1$
	$C0(a) = C0(z) + CC1(b) + 1$ $C0(b) = C0(z) + CC1(a) + 1$
	$C0(a) = C0(z) + CC0(b) + 1$ $C0(b) = C0(z) + CC0(a) + 1$
	$C0(a) = C0(z) + \min(CC0(b), CC1(b)) + 1$ $C0(b) = C0(z) + \min(CC0(a), CC1(a)) + 1$
	$C0(a) = C0(z) + 1$

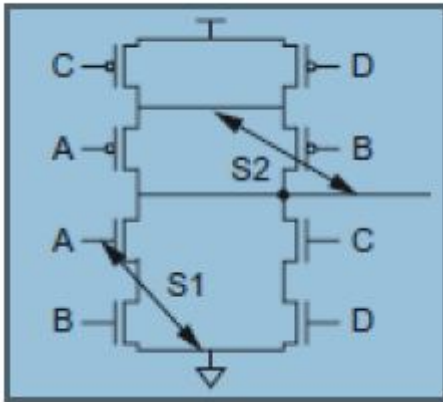


MANUFACTURING TEST PRINCIPLES:

STUCK AT Faults These faults occur when a node is accidentally connected to the power supply(SA1) or ground (SA0)

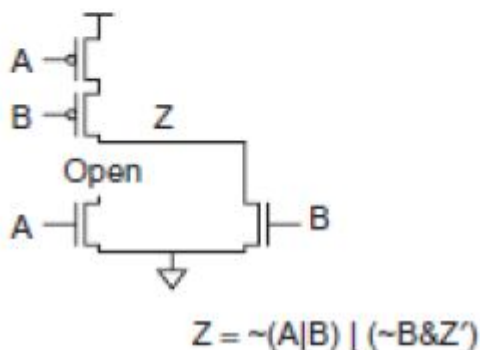
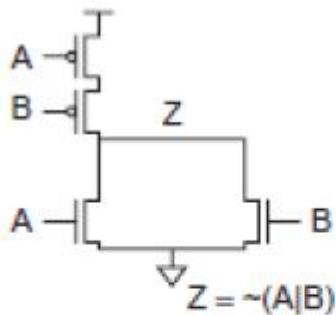


Short circuit and open circuit fault The short S1 results in an S-A-0 fault at input A, while short S2 modifies the function of the gate.



2-input NOR gate in which one of the transistors is rendered ineffective. If nMOS transistor A is stuck open, then the function displayed by the gate will be

$$Z = \overline{A} + \overline{B} + \overline{B}Z'$$



Fault Coverage

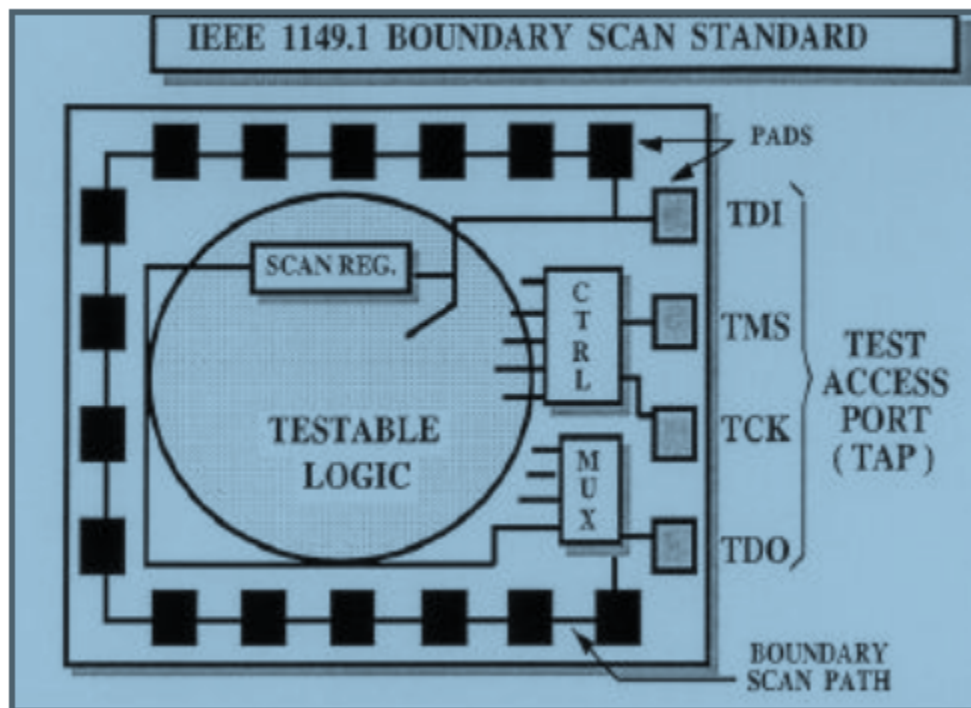
A measure of goodness of a set of test vectors is the amount of *fault coverage* it achieves. Each circuit node is taken in sequence and held to 0 (S-A-0), and the circuit is simulated with the test vectors comparing the chip outputs with a *known good machine*—a circuit with no nodes artificially set to 0 (or 1). When a discrepancy is detected between the *faulty machine* and the good machine, the fault is marked as detected and the simulation is stopped. This is repeated for setting

the node to 1 (S-A-1). In turn, every node is stuck (artificially) at 1 and 0 sequentially. The fault coverage of a set of test vectors is the percentage of the total nodes that can be detected as faulty when the vectors are applied.

Boundary Scan Test (BST)

Boundary Scan Test (BST) is a technique involving scan path and self-testing techniques to resolve the problem of testing boards carrying VLSI integrated circuits and/or surface mounted devices (SMD). Printed circuit boards (PCB) are becoming very dense and complex, especially with SMD circuits, that most test equipment cannot guarantee good fault coverage. BST (figure 8.15) consists in placing a scan path (shift register) adjacent to each component pin and to interconnect the cells in order to form a chain around the border of the circuit. The BST circuits contained on one board are then connected together to form a single path through the board. The boundary scan path is provided with serial input and output pads and appropriate clock pads which make it possible to:

- Test the interconnections between the various chip
- Deliver test data to the chips on board for self-testing
- Test the chips themselves with internal self-test



The advantages of Boundary scan techniques are as follows :

- No need for complex testers in PCB testing
- Test engineers work is simplified and more efficient
- Time to spend on test pattern generation and application is reduced
- Fault coverage is greatly increased.

LOGICAL VERIFICATION:

1. Validation via Simulation

Validation of the initial HDL description is a major bottleneck in the design process. Because the HDL description is usually the first description of the design, simulation is the primary methodology for validating it. Simulation based validation, however, is necessarily incomplete because it is not computationally feasible to exhaustively simulate the entire design. It is therefore important to measure the degree of verification coverage, both qualitatively and quantitatively. Coverage measures such as code (line) coverage, branch coverage, transition coverage, etc., have been used extensively in industry. A “high” degree of simulation coverage provides confidence to the designer regarding the correctness of the design. A “low” degree of coverage implies that certain functionalities of the design may not have been simulated. In which case, should there be an error in the non-simulated portion of the design, it may not have been observed during the validation process. It is therefore important to simulate the design functionality as much as possible. This requires automatic generation of a large number of simulation vectors to excite all the components in the design. Pseudo-random techniques for automatically generating simulation vectors are widely in use. Simulation tools may use random number generators, and randomly generate 1s and 0s, collect a vector of inputs and apply them to the design and perform the simulation. (Ever seen C language random number generators? Try reading the man-pages for `random()`, `randu()`, `drand48()`, etc. These are the most primitive random number generators....) While they do produce a virtually endless stream of simulation vectors, enhanced coverage is achieved only infrequently. Often, a situation arises where certain areas of code (or parts of the design) are not excited by the generated set of vectors. Validation tools can monitor every statement of code and keep a record of the number of times it was executed in response to the input stimulus. They can thus identify parts of the HDL design that have not been excited at all by the simulation vector set. After identifying the parts of the RTL description that were not excited by the simulation vectors, validation engineers have to generate the vectors that would excite these portions of the design.

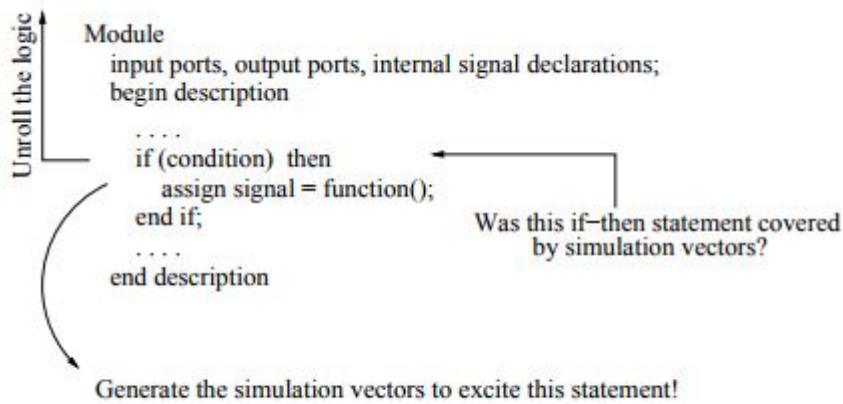


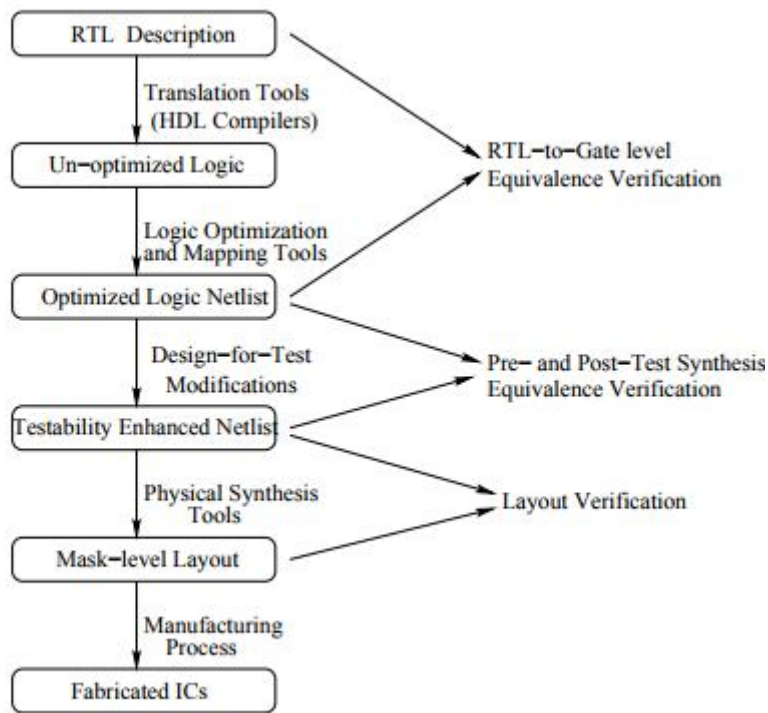
Fig. 3. RTL code coverage.

2. Formal Design Validation/verification

Design descriptions have traditionally been validated by extensive simulation. However, simulation based validation offers no guarantees for correctness. Because even if you have simulated the entire code - all if-then, case, for, while, etc., statements - you have NOT simulated (and CANNOT simulate) every permutation and combination of those if-then-else and 4 other statements. For this reason, verification (or should I say validation?) engineers increasingly rely upon mathematical techniques to formally prove the correctness of designs. Any digital system can be described mathematically, perhaps by a set of mathematical equations - in our case, by Boolean equations. A set of rules are used to generate formulas describing the system, and semantic functions are used that assign some meaning to the formulas. But what are you verifying (or validating)? You wish to verify certain properties of the system. For example, suppose that you are asked to design a traffic light controller for a 4-way intersection. You want to ensure, over the entire design space of the controller, that the traffic lights controlling two perpendicularly intersecting streets should never be green at the same time. Otherwise there will be an accident! If g_1 and g_2 are the green signals for the intersecting streets, your design should satisfy the property throughout the design space. In other words, we need a set of rules to generate formulas defining the properties that are to be checked, and semantic functions that define satisfiability of that property by the system. Now that we have a mathematical description of the system, as well as a mathematical representation of the properties that we wish to verify, all we need to do is formally verify whether or not our system satisfies the property. Since we are in the world of VLSI-CAD, we need an algorithm to verify the satisfaction relation. This is what is so dramatically called

“formal verification”. Traditional approaches to formal verification attempt to show that there exists a formal proof of the formulae defining the correctness criteria. The proof is obtained from the formulas characterizing the design and the axioms underlying the associated logical system. This process is referred to as theorem proving. The benefit of this approach is its generality and completeness. However, generating the proof automatically is cumbersome in both theory and practice. Existing heuristics to automatically generate the proof are both memory and compute intensive. As a result, theorem proving lacks the level of automation that is desirable for a CAD framework to be practically useful. Model checking provides a different approach to the formal verification problem. The system is characterized by a finite state transition graph (STG) where the vertices represent the configurations (or the states) that the system can reside in, and the edges represent the transition between the states. Properties of the system that are to be verified for satisfaction are represented using temporal logic formulae. Temporal logics are essentially equivalent to various special fragments of linear time logics or of branching time logics. Algorithmically, the verification is performed by traversing the state transition graph; starting from the initial state, the set of reachable states is found where all the states in the set satisfy the desired property. Model checking tools have achieved a significant level of automation and maturity and are widely in use in both academia and industry. One of the factors behind the success of model checking is that the STG of the underlying system can be relatively easily extracted from the designs described in either conventional hardware description languages or from circuit level netlists. Hence, the designers find it straightforward to include property verification within their design/synthesis methodology, as a common HDL-framework can be used for simulation, synthesis and verification.

3.Implementation Verification:



Typical implementation verification flow.

SILICON DEBUG PRINCIPLES:

Logic bugs vs. electrical failures

- Most chip failures are logic bugs from inadequate simulation
- Some are electrical failures
 - Crosstalk
 - Dynamic nodes: leakage, charge sharing
 - Ratio failures – A few are tool or methodology failures (e.g. DRC)

‰ Fix the bugs and fabricate a corrected chip

SHMOO PLOTS:

How to diagnose failures?

- Hard to access chips
 - Picoprobes

- Electron beam
- Laser voltage probing
- Built-in self-test %

Shmoo plots

- Vary voltage, frequency
- Look for cause of electrical failures

Test Pattern Generation:

Manufacturing test ideally would check every node in the circuit to prove it is not stuck. % Apply the smallest sequence of test vectors necessary to prove each node is not stuck. Good observability and controllability reduces number of test vectors required for manufacturing test. – Reduces the cost of testing – Motivates design-for-test

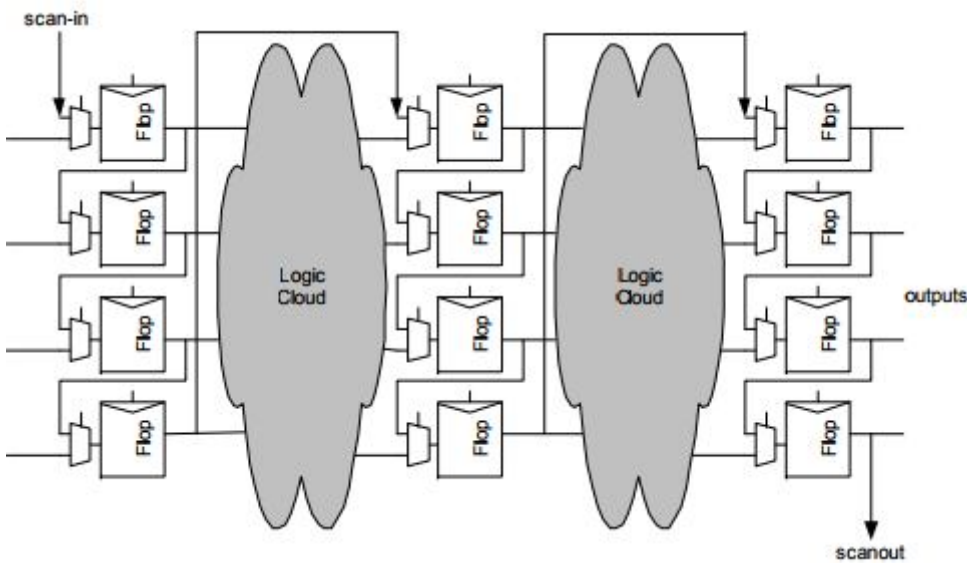
Design for Test

%

Design the chip to increase observability and controllability .If each register could be observed and controlled, test problem reduces to testing combinational logic between registers. Better yet, logic blocks could enter test mode where they generate test patterns and report the results automatically.

Scan:

Convert each flip-flop to a scan register – Only costs one extra multiplexer.In Normal mode: flip-flops behave as usual .In Scan mode: flip-flops behave as shift register and Contents of flops can be scanned out and new values scanned in



ATPG:

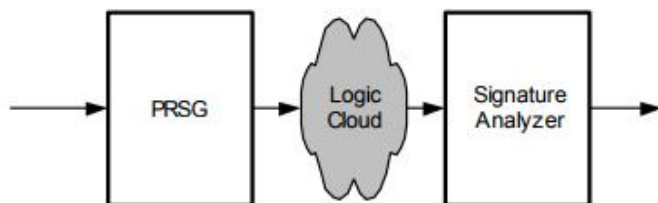
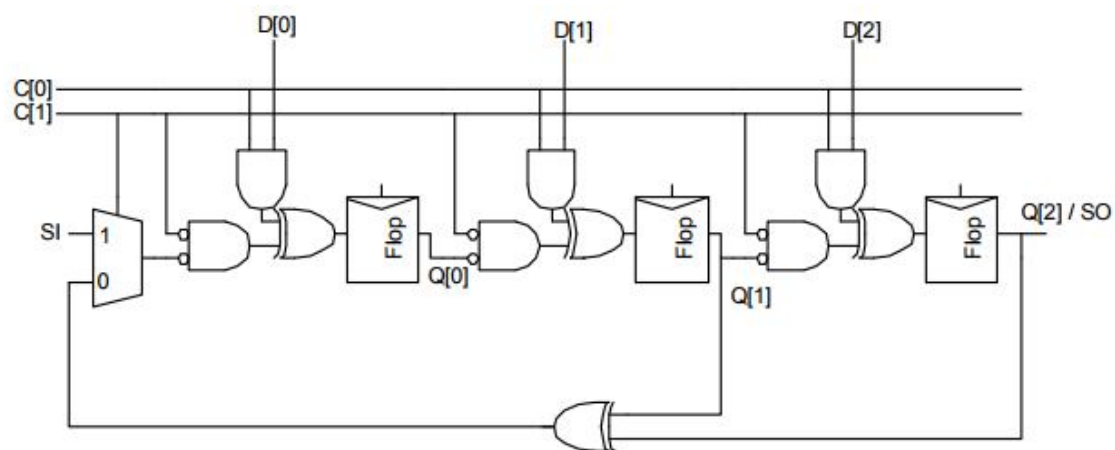
- Test pattern generation is tedious. %
- Automatic Test Pattern Generation (ATPG) tools produce a good set of vectors for each block of combinational logic %
- Scan chains are used to control and observe the blocks %
- Complete coverage requires a large number of vectors, raising the cost of test %
- Most products settle for covering 90+% of potential stuck-at faults

Built-in self-test:

- Built-in self-test lets blocks test themselves
- Generate pseudo-random inputs to comb. logic
- Combine outputs into a syndrome – With high probability, block is fault-free if it produces the expected syndrome

BILBO:

Built-in Logic Block Observer – Combine scan with PRSG & signature analysis



MODE	C[1]	C[0]
Scan	0	0
Test	0	1
Reset	1	0
Normal	1	1

UNIT V

SPECIFICATION USING VERILOG HDL

Introduction

With the advent of VLSI technology and increased usage of digital circuits, designers have 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, eventually 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 come 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 and 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: $x = \sim y$

\sim is a unary operator

y is the operand

binary : Comes between two operands.

Ex: $x = y \parallel z$

\parallel 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. Whitespaces 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

Unsize 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 used 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 vector 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 shown 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 posedge 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 be 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
-	Subtract	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, in0, 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 & in3);

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. Whereas 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
```

