EC64 VLSI DESIGN SYLLABUS

UNIT I CMOS TECHNOLOGY

A brief History-MOS transistor, Ideal I-V characteristics, C-V characteristics, Non ideal IV effects, DC transfer characteristics - CMOS technologies, Layout design Rules, CMOS process enhancements, Technology related CAD issues, Manufacturing issues

UNIT II CIRCUIT CHARACTERIZATION AND SIMULATION

Delay estimation, Logical effort and Transistor sizing, Power dissipation, Interconnect, Design margin, Reliability, Scaling- SPICE tutorial, Device models, Device characterization, Circuit characterization, Interconnect simulation

UNIT III COMBINATIONAL AND SEQUENTIAL CIRCUIT DESIGN

Circuit families –Low power logic design – comparison of circuit families – Sequencing static circuits, circuit design of latches and flip flops, Static sequencing element methodology- sequencing dynamic circuits – synchronizers

UNIT IV CMOS TESTING

Need for testing- Testers, Text fixtures and test programs- Logic verification- Silicon debug principles- Manufacturing test – Design for testability – Boundary scan

UNIT V SPECIFICATION USING VERILOG HDL

Basic concepts- identifiers- gate primitives, gate delays, operators, timing controls, procedural assignments conditional statements, Data flow and RTL, structural gate level switch level modeling, Design hierarchies, Behavioral and RTL modeling, Test benches, Structural gate level description of decoder, equality detector, comparator, priority encoder, half adder, full adder, Ripple carry adder, D latch and D flip flop.

Textbooks:

1. Weste and Harris: CMOS VLSI DESIGN (Third edition) Pearson Education, 2005
2. Uyemura J.P: Introduction to VLSI circuits and systems, Wiley 2002.
3. J.Bhasker,"Verilog HDl Primer" , BS publication,2001(UNIT V)

References:

1 D.A Pucknell & K.Eshraghian Basic VLSI Design, Third edition, PHI, 2003

2 Wayne Wolf, Modern VLSI design, Pearson Education, 2003

3 M.J.S.Smith: Application specific integrated circuits, Pearson Education, 1997

4 Ciletti Advanced Digital Design with the Verilog HDL, Prentice Hall of India, 2003

UNIT I CMOS TECHNOLOGY

INTRODUCTION

An MOS (Metal-Oxide-Silicon) structure is created by superimposing several layers of conducting, insulating, and transistor forming materials. After a series of processing steps, a typical structure might consists of levels called diffusion, polysilicon, and metal that are separated by insulating layers. CMOS technology provides two types of transistors, an n-type transistor (n MOS) and a p-type transistor (p MOS). These are fabricated in silicon by using either negatively doped silicon that is rich in electrons (negatively charged) or positively doped silicon that is rich in holes (the dual of electrons and positively charged). For the n-transistor, the structure consists of a section of p-type silicon separating two diffused areas of n-type silicon. The area separating the n regions is capped with a sandwich consisting of an insulator and a conducting electrode called the GATE. Similarly, for the p-transistor the structure consists of a section of n-type silicon separating two p-type diffused areas. The p-transistor also has a gate electrode. The gate is a control input and it affects the flow of electrical current between the drain and source. The drain and source may be viewed as two switched terminals.

An MOS transistor is termed a majority-carrier device, in which the current in a conducting channel between the source and drain is modulated by a voltage applied to the gate. In an n-type MOS transistor (i.e.,nMOS), the majority carriers are electrons. A positive voltage applied on the gate with respect to the substrate enhances the number of electrons in the channel (region immediately under the gate) and hence increases the conductivity of the channel. The operation of a p-type transistor is analogous to the nMOS transistor, with the exception that the majority carriers are holes and the voltages are negative with respect to the substrate. The switching behavior of an MOS device is characterized by threshold voltage, V_t. This is defined as the voltage at which an MOS device begins to conduct. For gate voltage less than a threshold value, the channel is cut-off, thus causing a very low drain- to-source current. Those devices that are normally cut-off (i.e., non-conducting) with zero gate bias are further classed as enhancement mode devices, whereas those devices that conduct with zero gate bias are called depletion mode devices.

MOS Transistor

So far, we have treated MOS transistors as ideal switches.

An *ON* transistor passes a finite amount of current

Depends on terminal voltages

Need to derive current-voltage (I-V) characteristics.

Transistor gate, source and drain all have capacitance

$$I = C(\Delta V / \Delta t)$$

$$\Delta t = (C / I) \cdot \Delta V$$

We will also look at what a *degraded level* really means.

Positive/negative voltage applied to the gate (with respect to substrate) enhances the number of electrons/holes in the channel and increases conductivity between source and drain.

V_t defines the voltage at which a MOS transistor begins to conduct. For voltages less than V_t (threshold voltage), the channel is cut off.

MOS Terminal Voltages and Modes of Operation

- Mode of operation depends on the terminal voltages. V_g, V_s, V_d
  - $V_{gs} = V_g - V_s$
  - $V_{gd} = V_g - V_d$
  - $V_{ds} = V_d - V_s = V_{gs} - V_{gd}$
- Source and Drain are symmetric diffusion terminals
  - By convention, source is terminal at lower voltage
  - Hence $V_{ds} > 0$
- NMOS body is grounded. First assume that source is 0 too.
- Three modes of operation
  - Cutoff
  - Linear
  - Saturation

MOS I-V Characteristics (Saturation, linear, cutoff)

MOS I-V characteristics (saturation)

If $V_{gd} < V_t$ channel pinches off near the drain

When $V_{ds} > V_{dsat} = V_{gs} - V_t$

Now drain voltage no longer increases current

$$I_{ds} = \beta\left((V_{gs} - V_t) - \frac{V_{dsat}}{2}\right)V_{dsat} = \frac{\beta}{2}(V_{gs} - V_t)^2$$

Shockley 1st order transistor models

$$I_{ds} = \begin{cases} 0 & V_{gs} < V_t & \textbf{Cutoff} \\ \beta\left((V_{gs} - V_t) - \frac{V_{ds}}{2}\right)V_{ds} & V_{ds} < V_{dsat} & \textbf{Linear} \\ \frac{\beta}{2}(V_{gs} - V_t)^2 & V_{ds} > V_{dsat} & \textbf{Saturation} \end{cases}$$

MOS I-V Characteristics (Saturation, linear, cutoff)

MOS I-V characteristics (saturation)

If $V_{gd} < V_t$ channel pinches off near the drain

When $V_{ds} > V_{dsat} = V_{gs} - V_t$

Now drain voltage no longer increases current

$$I_{ds} = \beta\left((V_{gs} - V_t) - \frac{V_{dsat}}{2}\right)V_{dsat} = \frac{\beta}{2}(V_{gs} - V_t)^2$$

Shockley 1st order transistor models

$$I_{ds} = \begin{cases} 0 & V_{gs} < V_t & \textbf{Cutoff} \\ \beta\left((V_{gs} - V_t) - \frac{V_{ds}}{2}\right)V_{ds} & V_{ds} < V_{dsat} & \textbf{Linear} \\ \frac{\beta}{2}(V_{gs} - V_t)^2 & V_{ds} > V_{dsat} & \textbf{Saturation} \end{cases}$$

Threshold Voltage

From equations, threshold voltage may be varied by changing:

- The doping concentration (N_A).
- The oxide capacitance (C_{ox}).
- Surface state charge (Q_{fc}).

As you can see, it is often necessary to adjust V_t. Two methods are common:

- Change Q_{fc} by introducing a small doped region at the oxide/substrate interface via ion implantation.

- Change C_{ox} by using a different insulating material for the gate.

A layer of Si_3N_4 (silicon nitride) with a relative permittivity of 7.5 is combined with a layer of silicon dioxide (relative permittivity of 3.9).
This results into a relative permittivity of 6.

For the same thickness dielectric layer, C_{ox} is larger using the combined material, which lowers V_t.

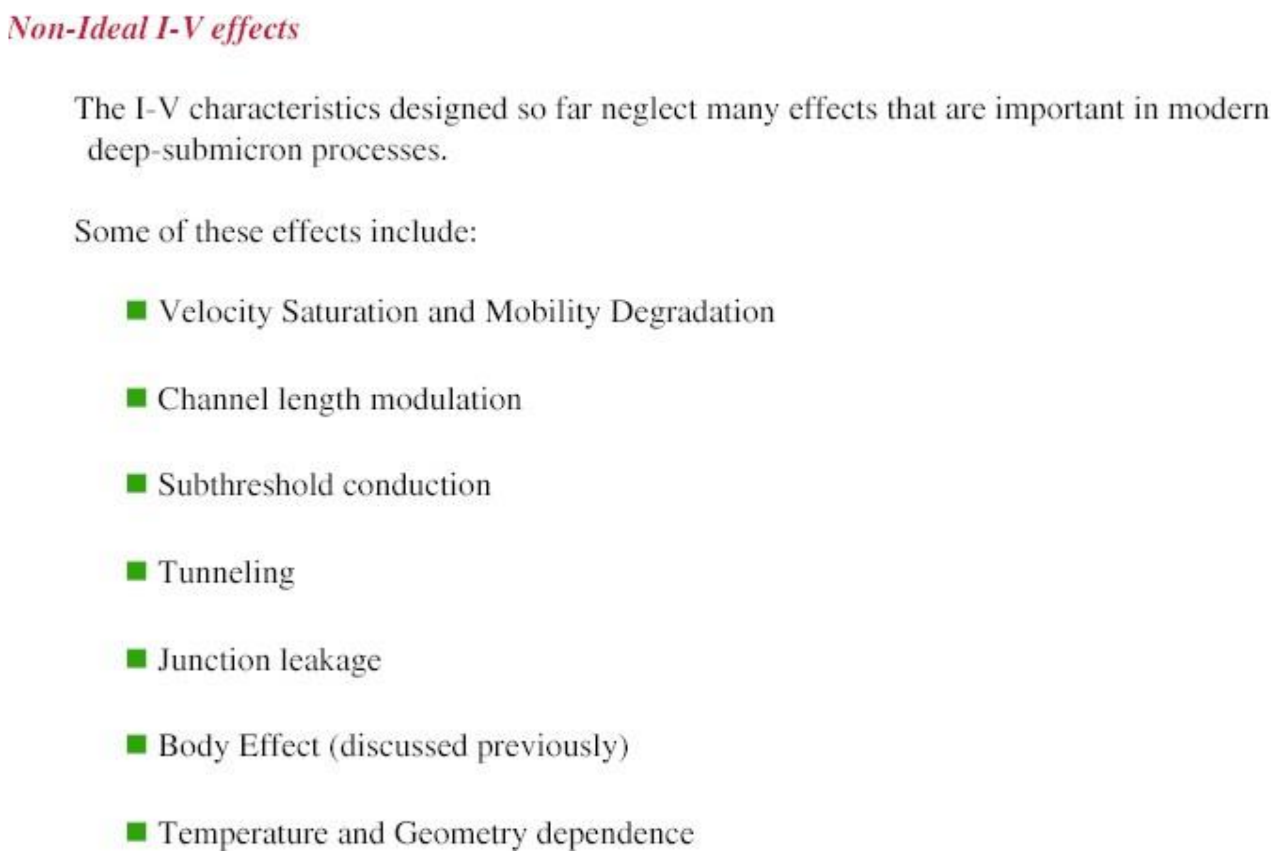
CV Characteristics:

The measured MOS capacitance (called gate capacitance) varies with the applied gate voltage

- A very powerful diagnostic tool for identifying any deviations from the ideal in both oxide and semiconductor
- Routinely monitored during MMOS device fabrication

Measurement of C-V characteristics

- Apply any dc bias, and superimpose a small (15 mV) ac signal
- Generally measured at 1 MHz (high frequency) or at variable frequencies between 1KHz to 1 MHz
- The dc bias V_G is slowly varied to get **quasi-continuous** C-V characteristics



Non-Ideal I-V effects

The I-V characteristics designed so far neglect many effects that are important in modern deep-submicron processes.

Some of these effects include:

- Velocity Saturation and Mobility Degradation
- Channel length modulation
- Subthreshold conduction
- Tunneling
- Junction leakage
- Body Effect (discussed previously)
- Temperature and Geometry dependence

DC Characteristics of CMOS inverter

The general arrangement and characteristics are illustrated in Fig. 2.3. The current/voltage relationships for the MOS transistor may be written as,

$$I_{ds} = K \frac{W}{L}\left((V_{ds} - V_t)V_{ds} - \frac{V_{ds}^2}{2}\right)$$

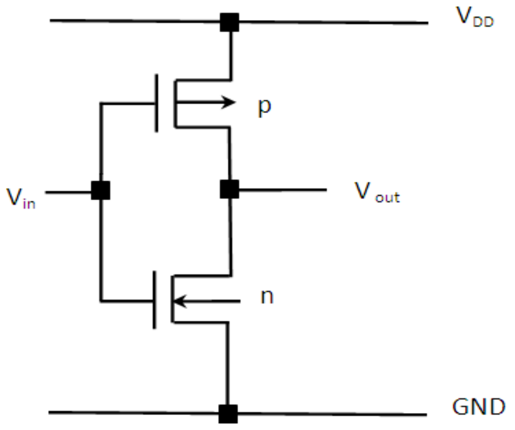


Figure 2.3 CMOS inverter

In the resistive region, or

$$I_{ds} = K \frac{W}{L} \frac{(V_{gs} - V_t)^2}{2}$$

In the saturation region. In both cases the factor K is a technology- dependent parameter such that

$$K = \frac{\varepsilon_{ins}\,\varepsilon_o\,\mu}{D}$$

The factor W/L is contributed by the geometry and it is common practice to write

$$\beta = K \frac{W}{L}$$

Such that,

$$I_{ds} = \beta \frac{(V_{gs} - V_t)^2}{2}$$

In saturation, and where β may be applied to both nMOS and pMOS transistors as follows,

$$\beta_n = \frac{\varepsilon_{ins}\,\varepsilon_o\,\mu_n}{D} \frac{W_n}{L_n}$$

$$\beta_p = \frac{\varepsilon_{ins}\,\varepsilon_o\,\mu_p}{D} \frac{W_p}{L_p}$$

Where W_n and L_n, W_p and L_p are the n- and p- transistor dimensions respectively. The CMOS inverter has five regions of operation is shown in Fig. 2.4 and in Fig. 2.5.

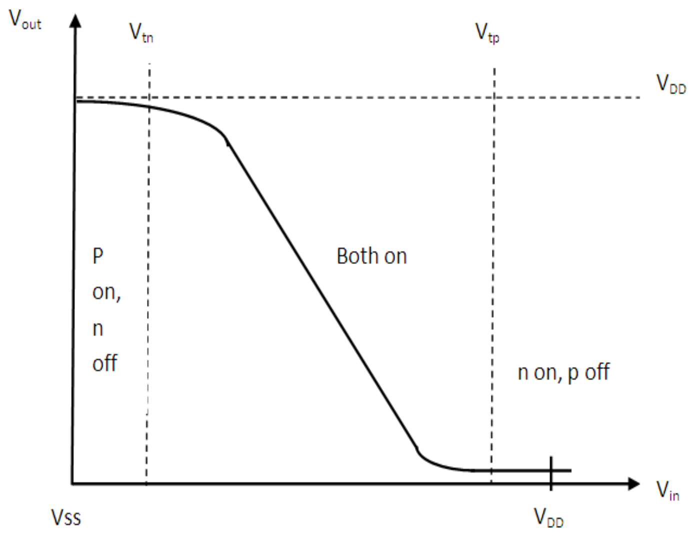


Figure 2.4 Transfer characteristics

Considering the static condition first, in region 1 for which V_{in} = logic 0, the p-transistor fully turned on while the n-transistor is fully turned off. Thus no current flows through the inverter and the output is directly connected to V_{DD} through the p-transistor.

In region 5 V_{in} = logic 1, the n-transistor is fully on while the p-transistor is fully off. Again, no current flows and a good logic 0 appears at the output.

In region 2 the input voltage has increased to a level which just exceeds the threshold voltage of the n-transistor. The n-transistor conducts and has a large voltage between source and drain. The p-transistor also conducting but with only a small voltage across it, it operates in the unsaturated resistive region.

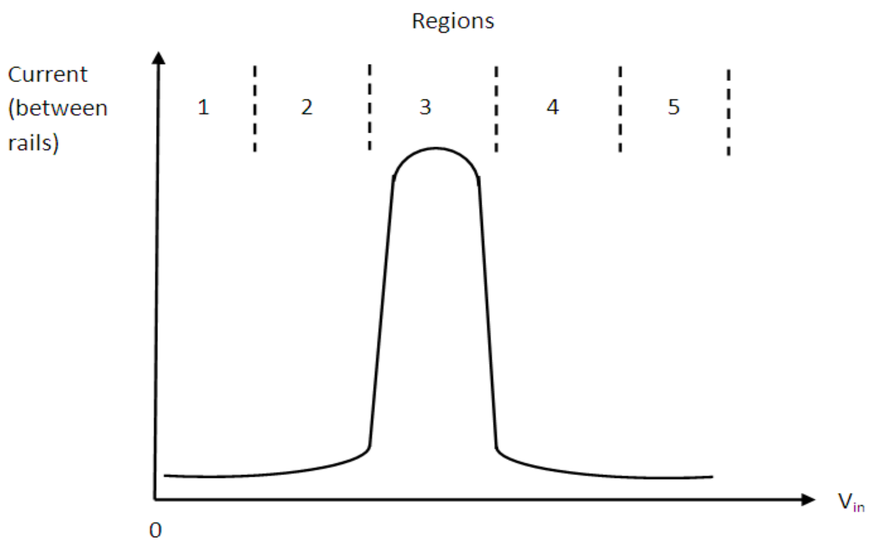


Figure 2.5 CMOS inverter current versus V_{in}

In region 4 is similar to region 2 but with the roles of the p- and n- transistors reversed. The current magnitudes in region 2 and 4 are small and most of the energy consumed in switching from one state to the other is due to the large current which flows in region 3.

In region 3 is the region in which the inverter exhibits gain and in which both transistors are in saturation.

The currents in each device must be the same since the transistors are in series. So we may write

$I_{dsp} = - I_{dsn}$

Where

$$I_{dsp} = \frac{\beta_n}{2} (V_{in} - V_{DD} - V_{tp})^2$$

And

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

V_{in} in terms of the β ratio and the other circuit voltages and currents

$$V_{in} = \frac{V_{DD} + V_{tp} + V_{tn} (\beta_n + \beta_p)^{1/2}}{1 + (\beta_n + \beta_p)^{1/2}}$$

Since both transistors are in saturation, they act as current sources so that the equivalent circuit in this region is two current sources so that the equivalent circuit in this region is two current sources in series between V_{DD} and V_{SS} with the output voltage coming from their common point. The region is inherently unstable in consequence and the change over from one logic level to the other is rapid.

If $\beta_n = \beta_p$ and if $V_{in} = -V_{tp}$, then

$$V_{in} = 0.5\ V_{DD}$$

Since only at this point will the two β factors be equal. But for $\beta_n = \beta_p$ the device geometries must be such that

$\mu_p W_p/L_p = \mu_n W_n/L_n$

The mobilities are inherently unequal and thus it is necessary for the width to length ratio of the p-device to be three times that of the n-device, namely

$$W_p/L_p = 2.5\ W_n/L_n$$

The mobility μ is affected by the transverse electric field in the channel and is thus independent on V_{gs}. It has been shown empirically that the actual mobility is

$$\mu = \mu_z (1 - Ø (V_{gs} - V_t)^{-1}$$

Ø is a constant approximately equal to 0.05 V_t includes anybody effect, and μz is the mobility with zero transverse field.

CMOS Technologies

Classification of Silicon Technology

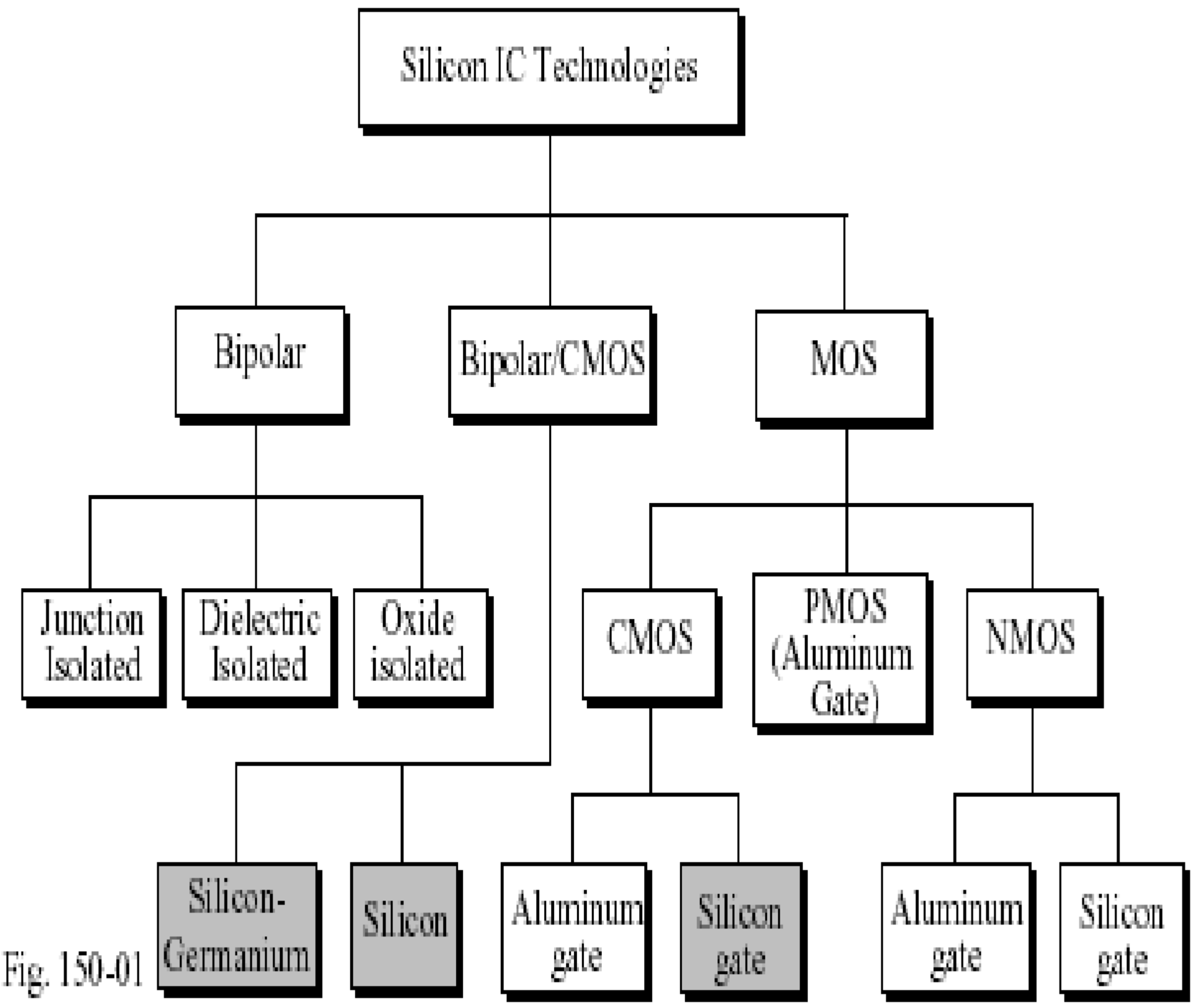


Fig. 150-01

VLSI DESIGN RULES

(From *Physical Design of CMOS Integrated Circuits Using L-EDIT*, John P. Uyemura)

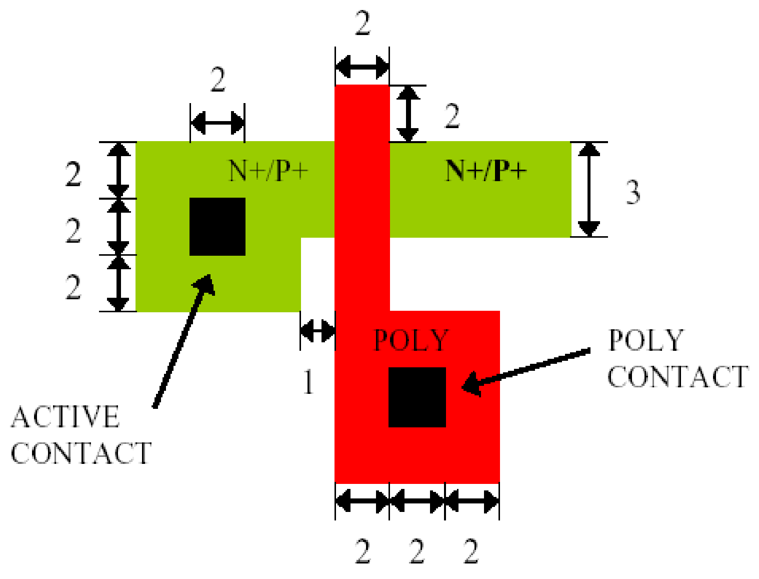
$\lambda = 1\mu m$

MINIMUM WIDTH AND SPACING RULES

LAYER	TYPE OF RULE	VALUE
POLY	Minimum Width Minimum Spacing	2λ 2λ
ACTIVE	Minimum Width Minimum Spacing	3λ 3λ
NSELECT	Minimum Width Minimum Spacing	3λ 3λ
PSELECT	Minimum Width Minimum Spacing	3λ 3λ
METAL1	Minimum Width Minimum Spacing	3λ 3λ

MOSFET LAYOUT RULES

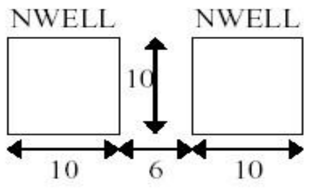
RULE	MEANING	VALUE
POLY Overlap	Minimum extension over ACTIVE	2λ
POLY-ACTIVE	Minimum Spacing	1λ
MOSFET Width	Minimum N+/P+ MOSFET W	3λ
ACTIVE CONTACT	Exact Size Minimum Space to ACTIVE Edge	2λ x 2λ 2λ
POLY CONTACT	Exact Size Minimum Space to POLY Edge	2λ x 2λ 2λ



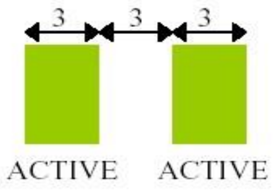
SCNA DESIGN RULE SET

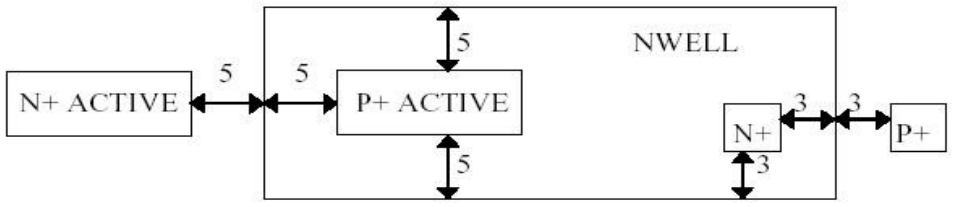
(**S**calable **C**MOS **N**-well **A**nalog)

1.0 NWELL	
1.1 Minimum Width.	10
1.3 Minimum Spacing	6



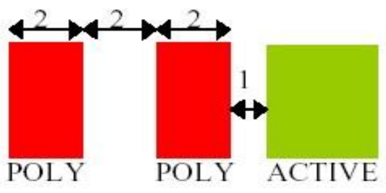
2.0 ACTIVE (N+, P+)	
2.1 Minimum Width.	3
2.2 Minimum Spacing	3
2.3 Drain/Source ACTIVE to NWELL	
2.3a P+ ACTIVE to NWELL . .	5
2.3b N+ ACTIVE to NWELL. .	5
2.4 CONTACT to NWELL EDGE	
2.4a P+ in SUB to NWELL. . .	3
2.4b N+ in WELL to NWELL. .	3

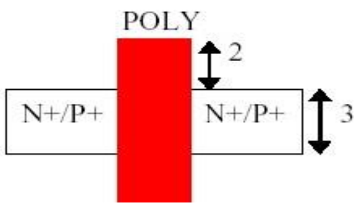




N+ = (NSELECT) AND (ACTIVE)
P+ = (PSELECT) AND (ACTIVE)

3.0 POLY	
3.1 Minimum Width.	2
3.2 Minimum Spacing	2
3.3 Gate Extension out of ACTIVE. .	2
3.4 Extension (MOSFET)	
3.4a nMOSFET Drain/Source . .	3
3.4b pMOSFET Drain/Source . .	3
3.5 POLY to ACTIVE Spacing . . .	1

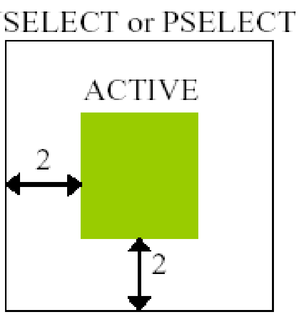




N+ = (NSELECT) AND (ACTIVE)
P+ = (PSELECT) AND (ACTIVE)

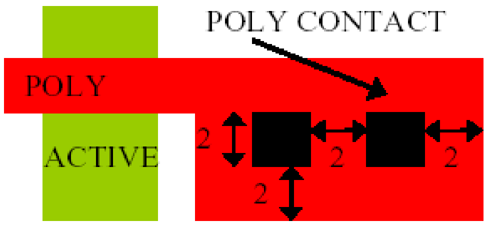
4.0 NSELECT and PSELECT

4.2 ACTIVE - SELECT Spacing	
4.2a ACTIVE in SELECT	2
4.2b ACTIVE in SELECT to ACTIVE in next SELECT .	2
4.4 Minimum Dimensions	
4.4a NSELECT Minimum Width	2
4.4b PSELECT Minimum Width	2
4.4c NSELeCT Minimum Space	2
4.4d PSELECT Minimum Space	2
4.5 PSELECT overlap of NSELECT .	0



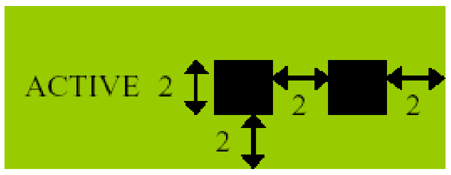
5.0 POLY CONTACT

5.1 Exact Size	2 x 2
5.2 Field Poly Overlap of POLY CONTACT.	2
5.3 Spacing	2



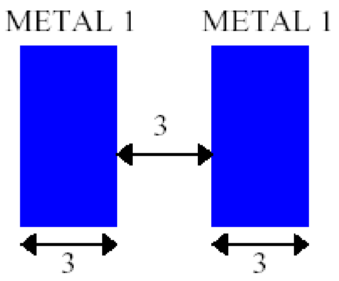
6.0 ACTIVE CONTACT

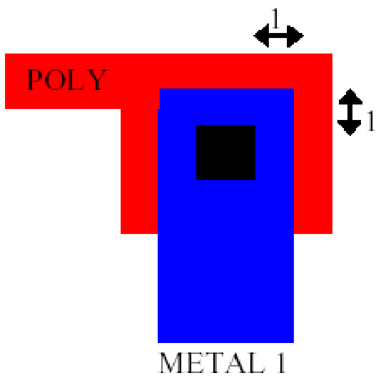
6.1 Minimum Width.	2
6.2 Field Poly Overlap of POLY CONTACT.	2
6.3 Spacing	2

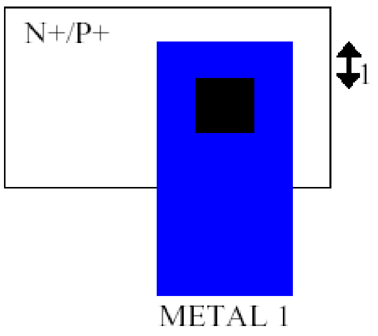


7.0 METAL 1

7.1 Minimum Width.	3
7.2 Minimum Spacing	3
7.3 Overlap of POLY CONTACT . .	1
7.4 Overlap of ACTIVE CONTACT .	1

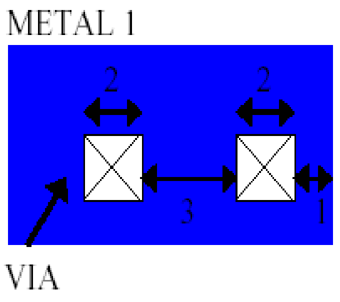






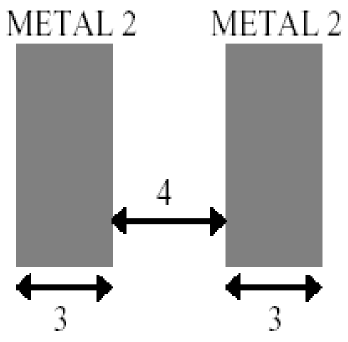
8.0 VIA

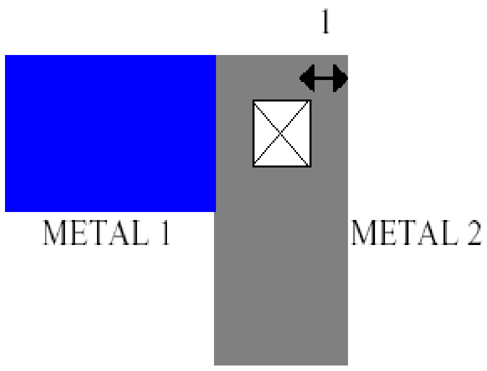
8.1 Exact Size	2 x 2
8.2 VIA to VIA Spacing	3
8.3 METAL 1 Overlap of VIA	1
8.4 VIA Spacing	
8.4a VIA to POLY	2
8.4b VIA (on POLY) to POLY	2
8.4c VIA to ACTIVE	2
8.4d VIA (on ACTIVE) to POLY	2



9.0 METAL 2

9.1 Minimum Width	3
9.2 Minimum Spacing	4
9.3 Overlap of VIA	4





CMOS TECHNOLOGIES

CMOS provides an inherently low power static circuit technology that has the capability of providing a lower-delay product than comparable design-rule nMOS or pMOS technologies. The four dominant CMOS technologies are:

- P-well process
- n-well process
- twin-tub process
- Silicon on chip process

The p-well process

A common approach to p-well CMOS fabrication is to start with moderately doped n-type substrate (wafer), create the p-type well for the n-channel devices, and build the p-channel transistor in the native n-substrate. The processing steps are,

1. The first mask defines the p-well (p-tub) n-channel transistors (Fig. 1.5a) will be fabricated in this well. Field oxide (FOX) is etched away to allow a deep diffusion.

2. The next mask is called the "thin oxide" or "thinox" mask (Fig. 1.5b), as it defines where areas of thin oxide are needed to implement transistor gates and allow implantation to form p- or n- type diffusions for transistor source/drain regions. The field oxide areas are etched to the silicon surface and then the thin oxide areas is grown on these areas. O ther terms for this mask include active area, island, and mesa.

3. Polysilicon gate definition is then completed. This involves covering the surface with polysilicon (Fig 1.5c) and then etching the required pattern (in this case an inverted "U"). "Poly" gate regions lead to "self-aligned" source-drain regions.

4. A p-plus (p^+) mask is then used to indicate those thin-oxide areas (and polysilicon) that are to be implanted p^+. Hence a thin-oxide area exposed by the p-plus mask (Fig. 1.5d) will become a p^+ diffusion area. If the p-plus area is in the n-substrate, then a p-channel transistor or p-type wire may be constructed. If the p-plus area is in the p-well, then an ohmic contact to the p-well may be constructed.

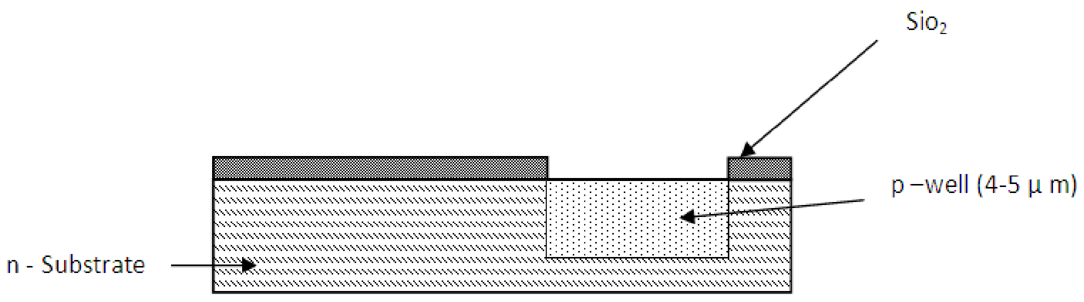
5. The next step usually uses the complement of the p-plus mask, although an extra mask is normally not needed. The "absence" of a p-plus region over a thin-oxide area indicates that the area will be an n^+ diffusion or n-thinox. n-thinox in the p-well defines possible n-

transistors and wires. An n^+ diffusion (Fig. 1.5e) in the n-substrate allows an ohmic contact to be made. Following this step, the surface of the chip is covered with a layer of Sio2.

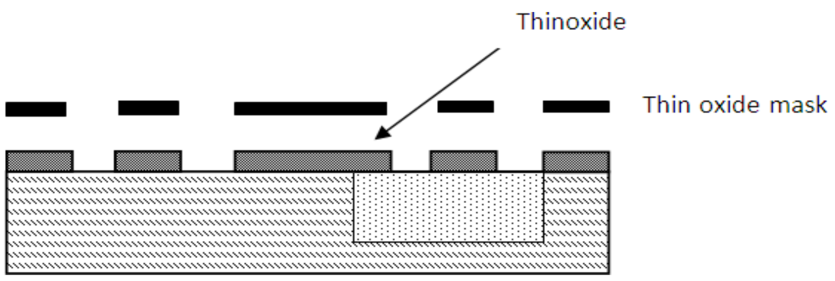
6. Contacts cuts are then defined. This involves etching any Sio_2 down to the contacted surface, these allow metal (Fig. 1.5f) to contact diffusion regions or polysilicon regions.

7. Metallization (Fig. 1.5g) is then applied to the surface and selectively etched.

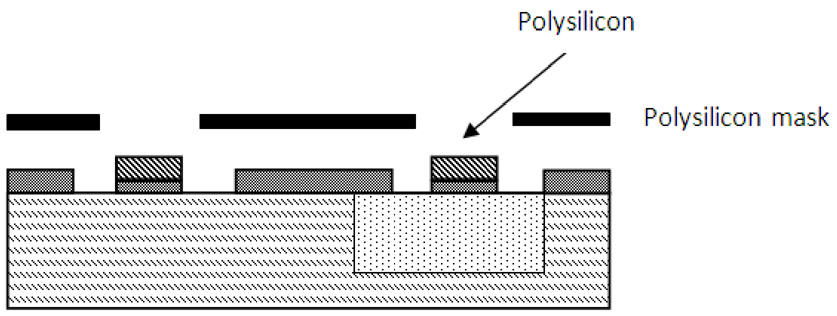
8. As a final step, the wafer is passivated and openings to the bond pads are etched to allow for wire bonding. Passivation protects the silicon surface against the ingress of contaminants.



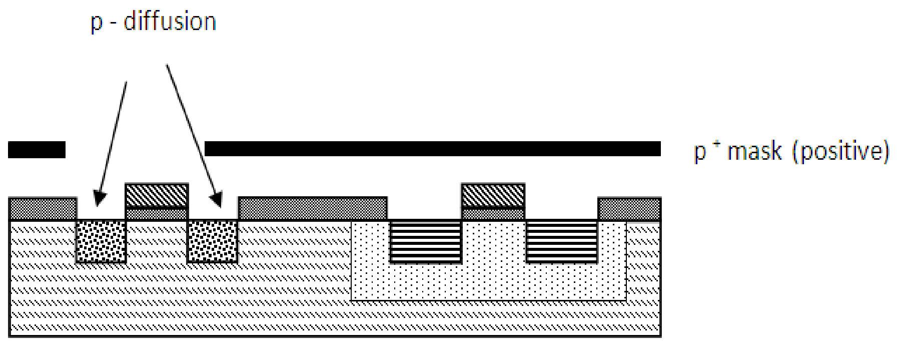
(a)



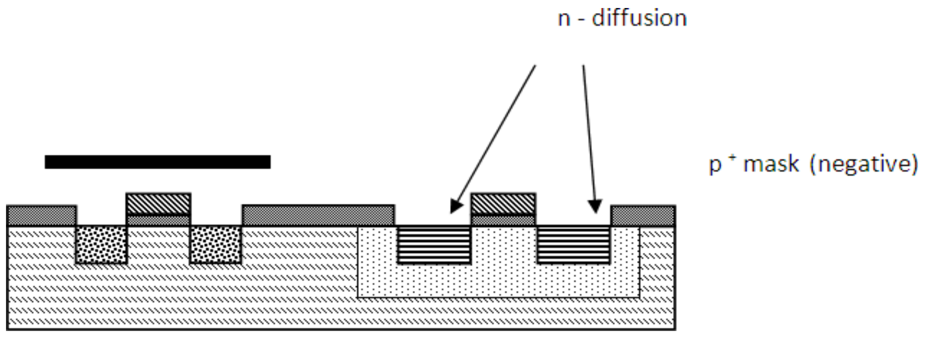
(b)



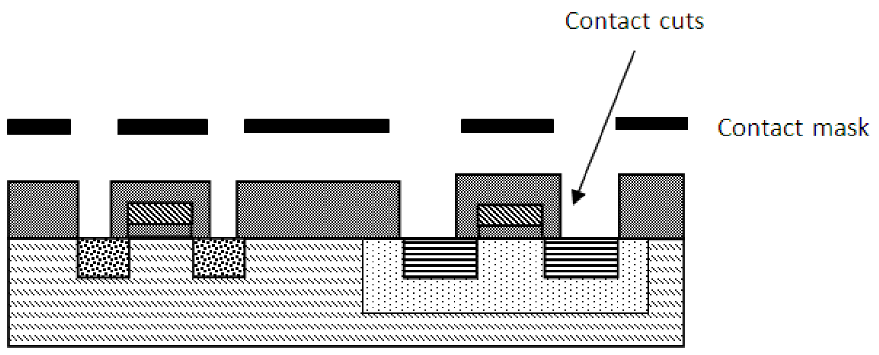
(c)



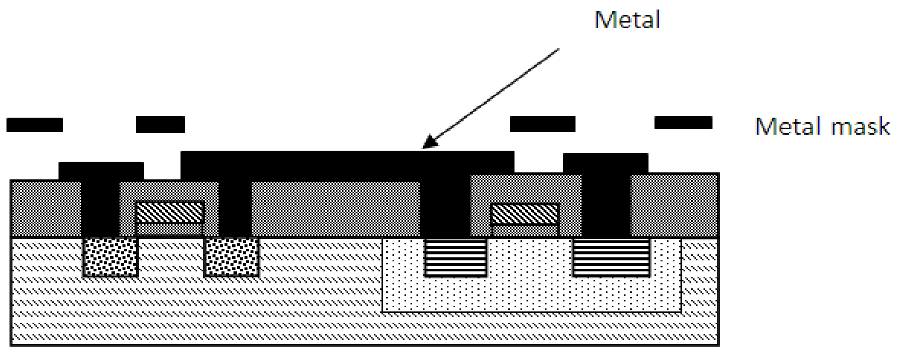
(d)



(e)



(f)



(g)

Figure 1.5 Typical p-well CMOS process steps with corresponding masks required

Basically the structure consists of an n-type substrate in which p-devices may be formed by suitable masking and diffusion and, in order to accommodate n-type devices, a deep p-well is diffused into the n-type substrate. This diffusion must be carried out with special care since the p-well doping concentration and depth will affect the threshold voltages as well as the breakdown voltages of the n-transistors. To achieve low threshold voltage (0.6 to 1.0 V), deep well diffusion or high well resistivity is needed. However, deep wells require larger spacing between the n- and p-type transistors and wires because of lateral diffusion resulting in larger chip areas.

High resistivity can accentuate latch-up problems. In order to achieve narrow threshold voltage tolerances in a typical p-well process, the well concentration is made about one order of magnitude higher than the substrate doping density, thereby causing the body effect for n-channel devices to be higher than for p-channel transistors. In addition, due to this higher concentration, n-transistors suffer from excessive source/drain to p-well capacitance will tends to be slower in performance. The well must be grounded in such a way as to minimize any voltage drop due to injected current in substrate that is collected by the p-well.

The p-well act as substrate for then-devices within the parent n-substrate, and, provided polarity restrictions are observed, the two areas are electrically isolated such that there are in affect two substrate, two substrate connections (V_{DD} and V_{SS}) are required.

The n-well process:

The p-well processes have been one of the most commonly available forms of CMOS. However, an advantage of the n-well process is that it can be fabricated on the same process line as conventional n MOS. n –well CMOS circuits are also superior to p-well because of the lower substrate bias effects on transistor threshold voltage and inherently lower parasitic capacitances associated with source and drain regions.

Typically n-well fabrication steps are similar to a p-well process, except that an n-well is used which is illustrated in Fig. 1.6. The first masking step defines the n-well regions. This followed by a low phosphorus implant driven in by a high temperature diffusion step to form the n-wells. The well depth is optimized to ensure against p-substrate to p^+ diffusion breakdown without compromising the n-well to n^+ mask separation. The next steps are to define the devices and diffusion paths, grow field oxide, deposit and pattern the polysilicon, carry out the diffusions, make contact cuts and metallization. An n-well mask is used to define n-well regions, as opposed to a p-well mask in a p-well process. An n-plus (n^+) mask may be used to define the n-channel transistors and V_{DD} contacts. Alternatively, we could use a p-plus mask to define the p-channel transistors, as the masks usually are the complement of each other.

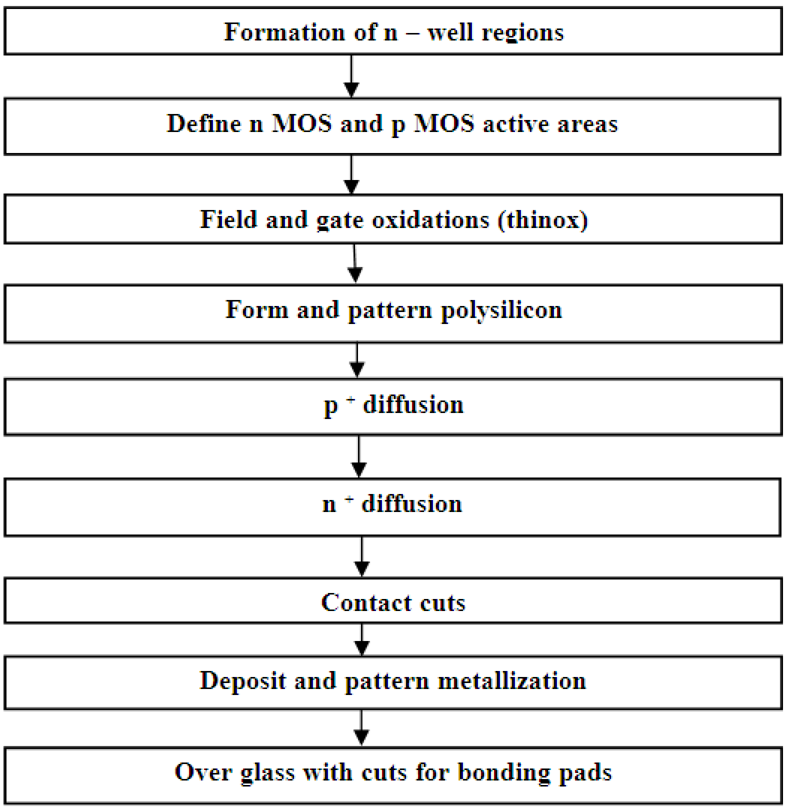

Formation of n – well regions
Define n MOS and p MOS active areas
Field and gate oxidations (thinox)
Form and pattern polysilicon
p + diffusion
n + diffusion
Contact cuts
Deposit and pattern metallization
Over glass with cuts for bonding pads

Figure 1.6 Main steps in a typical n-well process

Due to differences in mobility of charge carriers the n-well process creates non-optimum p-channel characteristics, such as high junction capacitance and high body effect. The n-well technology has a distinct advantage of providing optimum device characteristics. Thus n-channel devices may be used to form logic elements to provide speed and density, while p-transistors could primarily serve as pull-up devices.

The twin-tub process:

Twin-tub CMOS technology provides the basis for separate optimization of the p-type and n-type transistors, thus making it possible for threshold voltage, body effect, and the gain associated with n- and p-devices to be independently optimized. Generally the starting material is either an n^+ or p^+ substrate with a lightly doped epitaxial or epi layer, which is used for protection against latch-up. The aim of epitaxy is to grow high purity silicon layers of controlled thickness with accurately determined dopant concentrations distributed homogeneously throughout the layer. The electrical properties for this layer are determined by the dopant and its concentration in the silicon.

The process sequence, which is similar to the p-well process apart from the tub formation where both p-well and n-well are utilized as in Fig. 1.7, entails the following steps:

- Tub formation

- Thin oxide etching
- Source and drain implantations
- Contact cut definition
- Metallization.

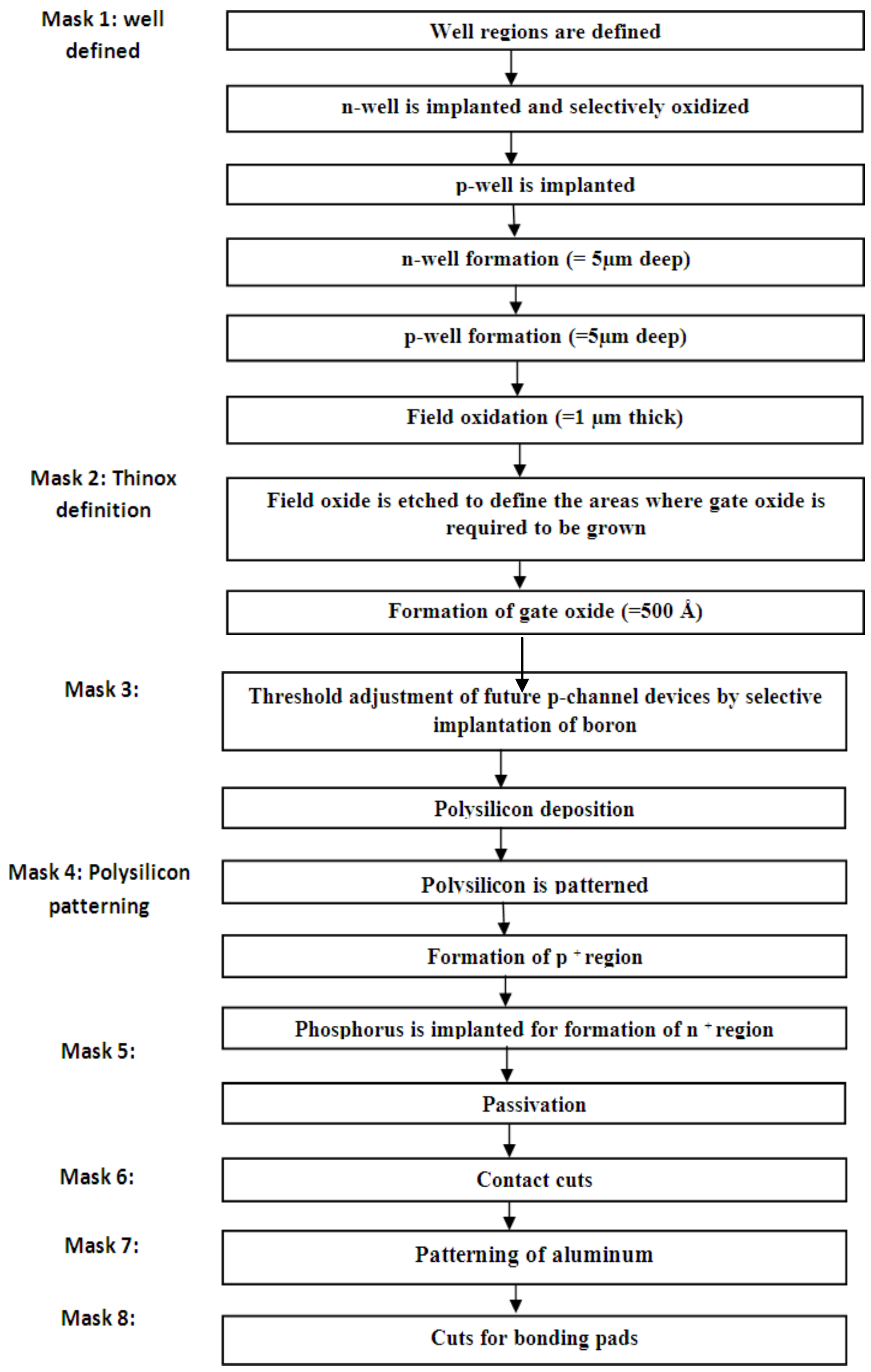

Mask 1: well defined
Well regions are defined
n-well is implanted and selectively oxidized
p-well is implanted
n-well formation (= 5μm deep)
p-well formation (=5μm deep)
Field oxidation (=1 μm thick)
Mask 2: Thinox definition
Field oxide is etched to define the areas where gate oxide is required to be grown
Formation of gate oxide (=500 Å)
Mask 3:
Threshold adjustment of future p-channel devices by selective implantation of boron
Polysilicon deposition
Mask 4: Polysilicon patterning
Polysilicon is patterned
Formation of p + region
Phosphorus is implanted for formation of n + region
Mask 5:
Passivation
Mask 6:
Contact cuts
Mask 7:
Patterning of aluminum
Mask 8:
Cuts for bonding pads

Figure 1.7 Flow diagram of twin-tub process

Since this process provides separately optimized wells, better performance n-transistors (lower capacitance, less body effect) may be constructed when compared with a conventional p-well process. Similarly the p-transistors may be optimized. The use of threshold adjust steps is included in this process.

Silicon on insulator process:

Silicon on insulator (SOI) CMOS processes has several potential advantages such as higher density, no latch-up problems, and lower parasitic capacitances. In the SOI process a thin layer of single crystal silicon film is epitaxial grown on an insulator such as sapphire or magnesium aluminate spinel. The steps involves are:

1) A thin film (7-8 μm) of very lightly doped n-type Si is grown over an insulator (Fig. 1.8a). Sapphire is a commonly used insulator.

2) An anisotropic etch is used to etch away the Si (Fig. 1.8b) except where a diffusion area will be needed.

3) The p-islands are formed next by masking the n-islands with a photoresist. A p-type dopant (boron) is then implanted. It is masked by the photoresist and at the unmasked islands. The p-islands (Fig. 1.8c) will become the n-channel devices.

4) The p-islands are then covered with a photoresist and an n-type dopant, phosphorus, is implanted to form the n-islands (Fig. 1.8d). The n-islands will become the p-channel devices.

5) A thin gate oxide (500-600Å) is grown over all of the Si structures (Fig. 1.8e). This is normally done by thermal oxidation.

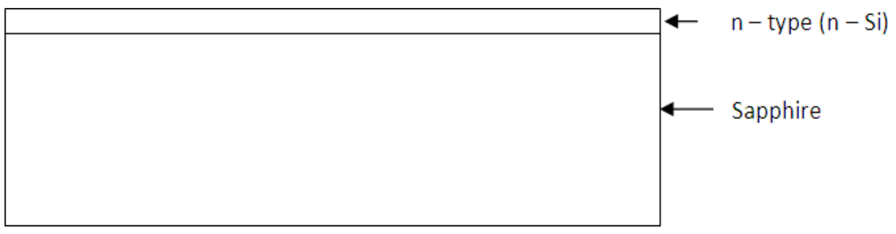
6) A polysilicon film is deposited over the oxide.

7) The polysilicon is then patterned by photomasking and is etched. This defines the polysilicon layer in the structure as in Fig. 1.8f.

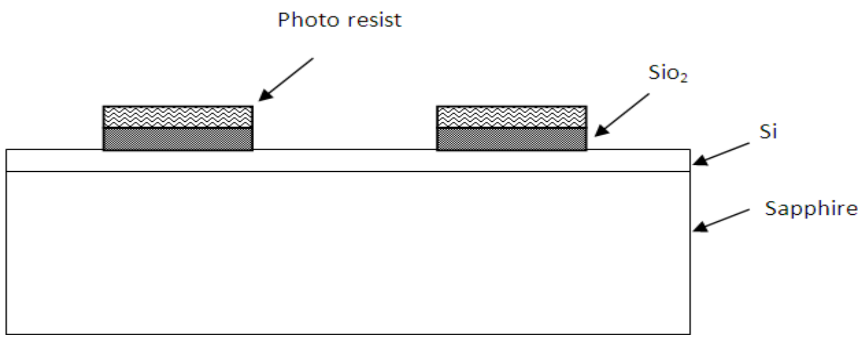
8) The next step is to form the n-doped source and drain of the n-channel devices in the p-islands. The n-island is covered with a photoresist and an n-type dopant (phosphorus) is implanted (Fig. 1.8g).

9) The p-channel devices are formed next by masking the p-islands and implanting a p-type dopant. The polysilicon over the gate of the n-islands will block the dopant from the gate, thus forming the p-channel devices is shown in Fig. 1.8h.

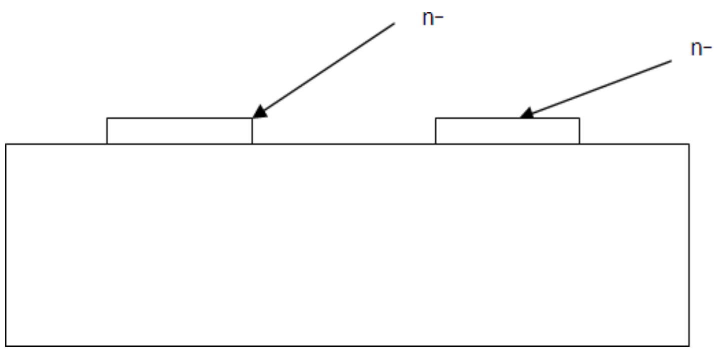
10) A layer of phosphorus glass is deposited over the entire structure. The glass is etched at contact cut locations. The metallization layer is formed. A final passivation layer of a phosphorus glass is deposited and etched over bonding pad locations.



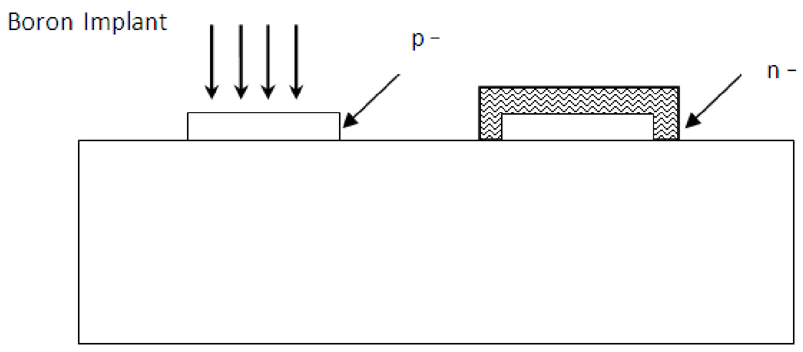
(a)



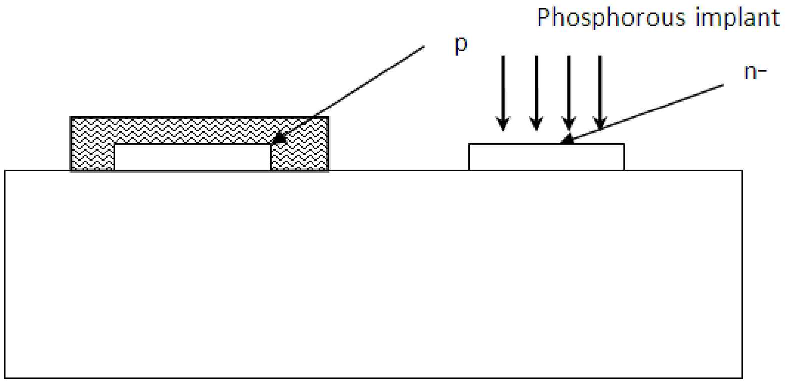
(b)



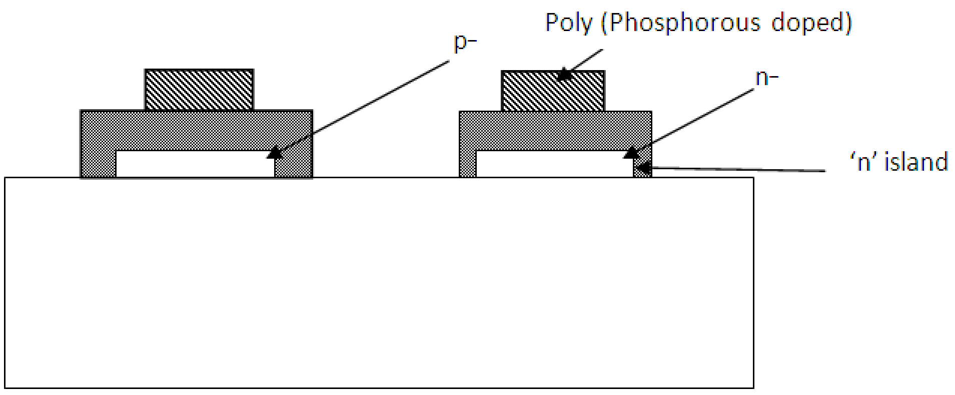
(c)



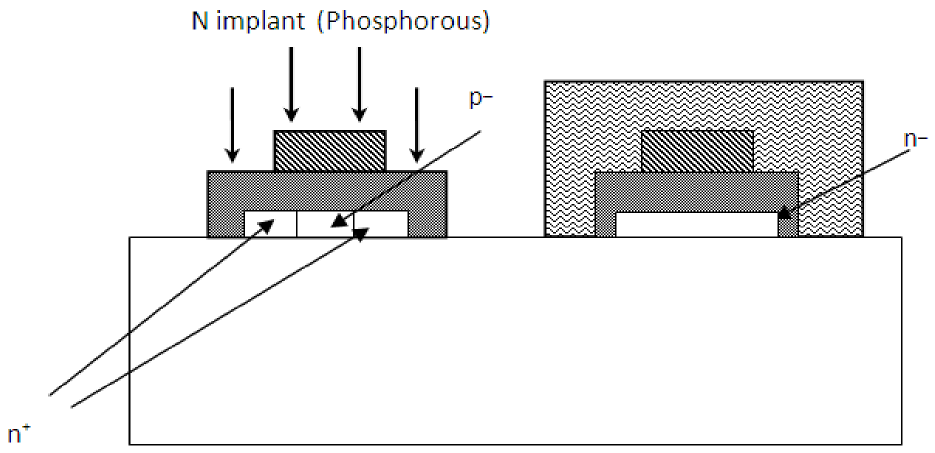
(d)


Phosphorous implant
p
n-

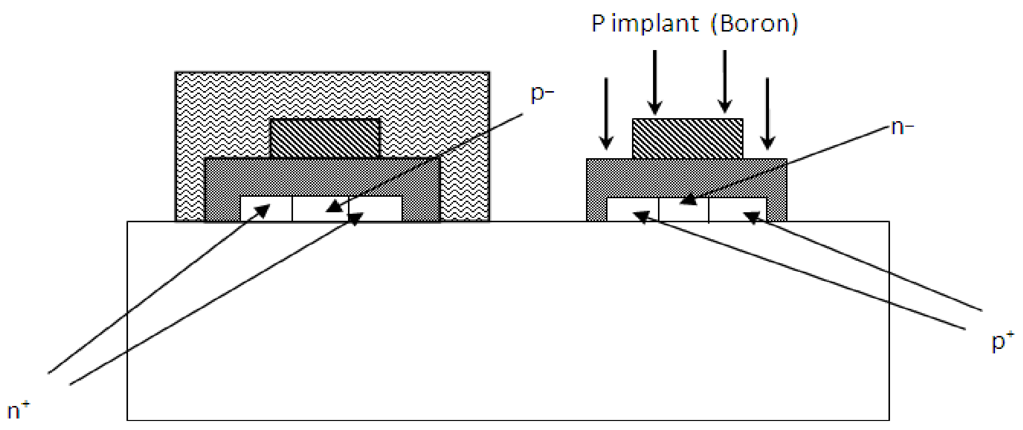
(e)


Poly (Phosphorous doped)
p-
n-
'n' island

(f)


N implant (Phosphorous)
p-
n-
n^+

(g)



(h)

Figure 1.8 SOI fabrication steps

The advantages of SOI technology are:

- Due to the absence of wells, denser structures than bulk silicon can be obtained.
- Low capacitances provide the basis of very fast circuits.
- No field-inversion problems exist.
- No latch-up due to isolation of n- and p- transistors by insulating substrate.
- As there is no conducting substrate, there are no body effect problems
- Enhanced radiation tolerance.

But the drawback is due to absence of substrate diodes, the inputs are difficult to protect. As device gains are lower, I/O structures have to be larger. Single crystal sapphires are more expensive than silicon and processing techniques tend to be less developed than bulk silicon techniques.

BiCMOS TECHNOLOGY FABRICATION

The MOS technology lies in the limited load driving capabilities of MOS transistors. This is due to the limited current sourcing and current sinking abilities associated with both p- and n- transistors. Bipolar transistors provide higher gain and have generally better noise and high frequency characteristics than MOS transistors and have effective way of speeding up VLSI circuits. When considering CMOS technology, there is difficulty in extending the fabrication processes to include bipolar as well as MOS transistors. Indeed, a problem of p-well and n-well CMOS processing is that parasitic bipolar transistors are formed as part of the outcome of fabrication.

The production of npn bipolar transistors with good performance characteristics can be achieved by extending the standard n-well CMOS processing to include further masks to add two additional layers such as the n^+ subcollector and p^+ base layers. The npn transistors is formed in an n-well and the additional p^+ base region is located in the well to form the p-base region of the transistor. The second additional layer, the buried n^+ subcollector (BCCD), is added to reduce the n-well (collector) resistance and thus improve the quality of the bipolar transistor. The arrangement of BiCMOS npn transistor is shown in Fig. 1.9.

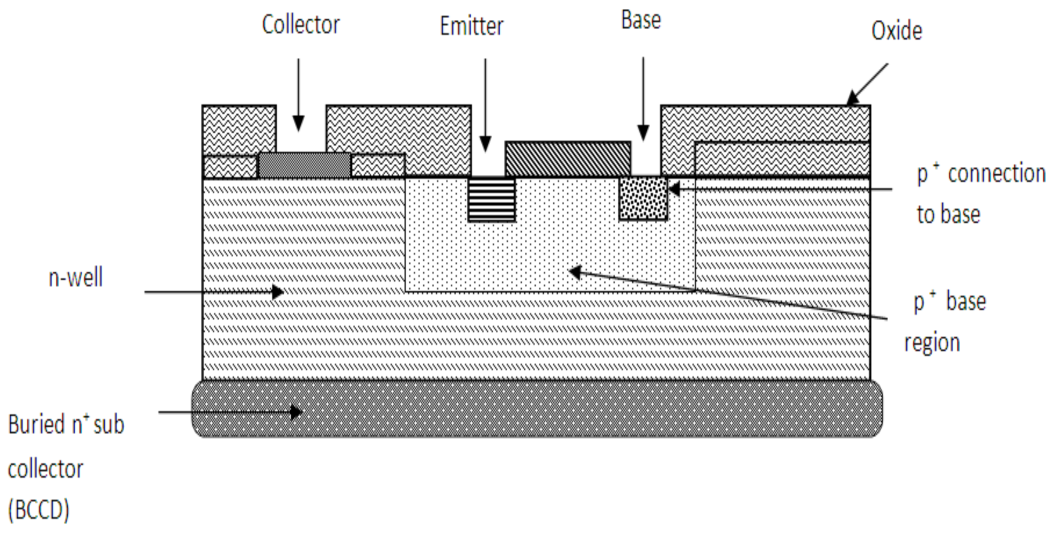


Figure 1.9 Arrangement of BiCMOS npn transistor

There are several advantages if the properties of CMOS and bipolar technologies could be combined. This is achieved to a significant extent in the BiCMOS technology. A further advantage which arises from BiCMOS technology is that analog amplifier design is facilitated and improved. High impedance CMOS transistors may be used for the input circuitry while the remaining stages and output drivers are realized using bipolar transistors. Since extra design and processing steps are involved as in Fig. 1.10, there is an increase in cost and some loss of packing density.

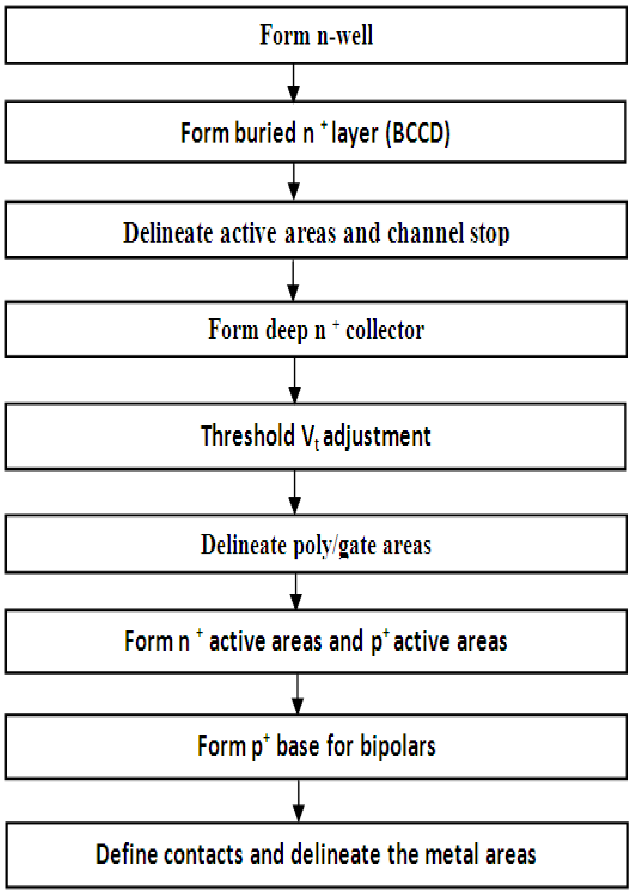

Form n-well
Form buried n + layer (BCCD)
Delineate active areas and channel stop
Form deep n + collector
Threshold V_t adjustment
Delineate poly/gate areas
Form n + active areas and p+ active areas
Form p+ base for bipolars
Define contacts and delineate the metal areas

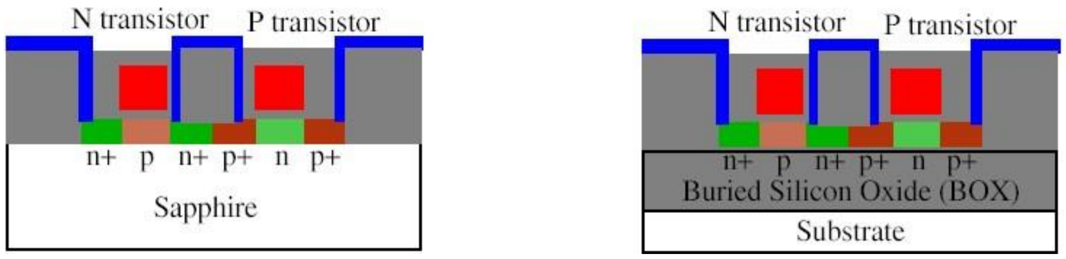
Figure 1.10 n-well BiCMOS fabrication steps

CMOS Process Enhancements

Transistors

- Multiple threshold voltages and oxide thicknesses
  - Processes offer multiple threshold voltages
  - Low threshold devices: faster, higher leakage. High threshold devices: opposite
  - Thin oxides: provide high ON currents but cannot handle high voltages (e.g. I/Os)
  - Thicker oxides provided for I/O devices

- Silicon on Insulator
  - As the name suggests transistors are fabricated on an insulator (SiO_2 or sapphire)
  - Insulating substrate eliminates capacitance between the source/drain and body, higher speed devices and low leakage currents.



- High-k gate dielectrics

  Transistors need high gate capacitance to attract charge to the channel

  Thin gates and therefore high gate leakages

  Thicker gates that leak less can be made with high-k materials

  e.g. hafnium oxide (k=20), zirconium oxide (k=23), silicon nitride (k=6.5-7.5)

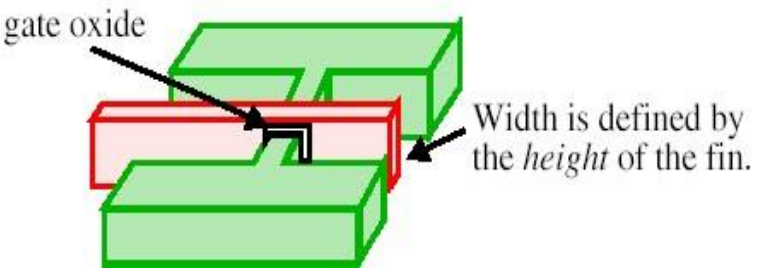
  Applied using ALD, MOCVD (metallo-organic CVD) or sputtering.

- Low leakage transistors

  Scaling transistors causes exponential increase in subthreshold leakages

  Can be improved using gate structure where gates is placed on more than one side of the channel

These devices are generically called ***finfets***



- Higher mobility

  Achieved by using SiGe (silicon germanium) for bipolar transistors in the same conventional CMOS process

  Silicon Germanium can also be used to improve speed by creating ***strained silicon***

- Plastic transistors

  MOS transistors fabricated with organic chemicals

  Used only for very specific applications as devices are very inexpensive to manufacture

- High-voltage transistors

  High voltage MOSFETs can be integrated onto conventional CMOS processes for switching and high-power applications.

  Specialized process steps required to achieve very high breakdown voltages.

Interconnect

- Copper Damascene process

  Using copper (higher conductivity) as interconnect instead of aluminum.

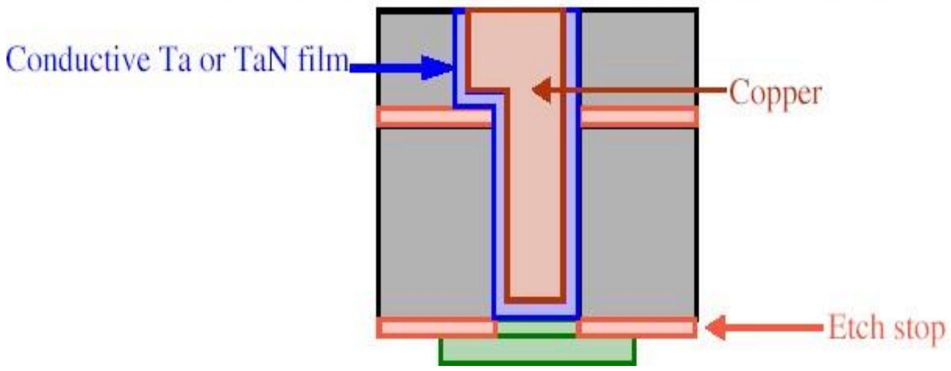
  Several challenges due to copper atom diffusion, etching issues and copper oxide formation.

  Special *barrier layers* used to prevent copper from entering the silicon surface.

  New metallization process called ***Damascene process*** invented to form barrier.

  Subtractive aluminum based metallization step involved adding aluminum everywhere and then etching it away leaving behind wires.

  Copper metallization is additive, trenches are made and then filled with copper.



- Low-k Dielectrics

  Low-k dielectrics between wires are attractive as they decrease wire capacitance.

  Reduces wire delay and power consumption.

  130nm process uses fluorosilicate glass (FSG, flourine added to silicon dioxide).

  Other materials have been developed, active research area.

Circuit Elements

Increasing use of CMOS for mixed signal and RF designs, requires special circuit elements with good characteristics

- Capacitors

  In conventional CMOS, capacitors can be created using the gate and source/drain, a diffusion area (to ground or V_{DD}) or a parallel metal plate capacitor.

  Enhancements include addition of a second polysilicon layer.

  Other enhanced types include metal-insulator-metal (MiM) capacitor and fringe (fractal) capacitor.

- Resistors

  In conventional CMOS, resistors can be built from any layer, where the final resistance depends on the resistivity (resistance per unit area) of the layer.

  Large resistance in small areas built using poly or diffusion.

  However, poly is usually doped so undoped poly allowed in enhanced processes.

  Another enhancement that requires additional processing steps is to allow nichrome that produces high quality resistors.

- Inductors

  Most common monolithic inductor is the spiral inductor, which is a spiral of upper-level metal. Used mainly for RF designs.

  Several enhancements techniques used to increase Q (self-resonant frequency).

- Other elements

  Other enhancements allow transmission lines, non-volatile memory, bipolar transistors, fuses and antifuses and micro electro mechanical systems (MEMS).

UNIT II CIRCUIT CHARACTERIZATION AND SIMULATION

Delay estimation:

Estimation of the delay of a Boolean function from its functional description is an important step towards design exploration at the register transfer level (RTL). This paper addresses the problem of estimating the delay of certain optimal multi-level implementations of combinational circuits, given only their functional description. The proposed delay model uses a new complexity measure called the delay measure to estimate the delay. It has an advantage that it can be used to predict both, the minimum delay (associated with an optimum delay implementation) and the maximum delay (associated with an optimum area implementation) of a Boolean function without actually resorting to logic synthesis. The model is empirical and results demonstrating its feasibility and utility are presented.

tpdr: rising propagation delay

From input to rising output crossing VDD/2

tpdf: falling propagation delay

From input to falling output crossing VDD/2

tpd: average propagation delay

tpd = (tpdr + tpdf)/2

tr: rise time

From output crossing 20% to 80% VDD

tf: fall time

From output crossing 80% to 20% VDD

tcd: average contamination delay

tcd = (tcdr + tcdf)/2

tcdr: rising contamination delay: Min from input to rising output crossing VDD/2

tcdf: falling contamination delay: Min from input to falling output crossinVDD/2

Solving differential equations by hand is hard. SPICE like simulators used for accurate analysis. But simulations are expensive. We need to be able to estimate delay although not as accurately as simulator.

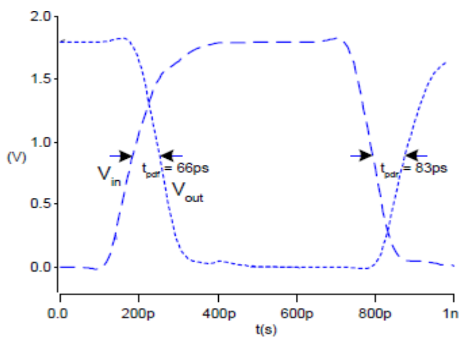
Use RC delay models to estimate delay

C = total capacitance on the output node

Use Effective resistance R

Therefore tpd = RC

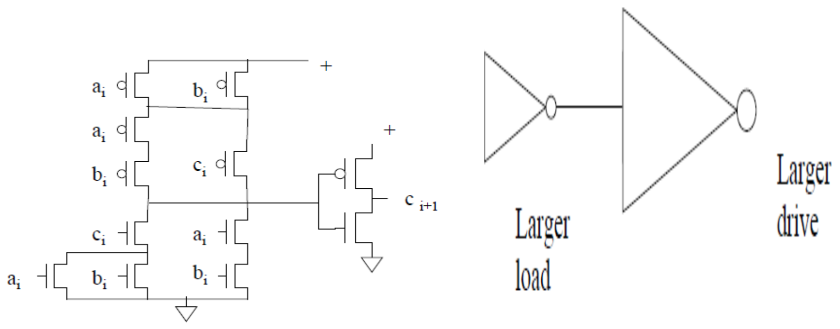
Transistors are characterized by finding their effective R.



Transistor sizing:

- Not all gates need to have the same delay.
- Not all inputs to a gate need to have the same delay.
- Adjust transistor sizes to achieve desired delay.

Example: Adder carry chain



Inter-stage effects in transistor sizing

□ Increasing a gate's drive also increases the load to the previous stage

Logical effort

Logical effort is a gate delay model that takes transistor sizes into account. Allows us to optimize transistor sizes over combinational networks. Isn't as accurate for circuits with reconvergent fanout.

Logical effort gate delay model

□ Express delays in process-independent unit

□ Gate delay is measured in units of minimum-size

$$d = \frac{d_{abs}}{\tau}$$

$$\begin{aligned} \tau &= 3RC \\ &\approx 12 \text{ ps in 180 nm process} \\ & 40 \text{ ps in 0.6 } \mu\text{m process} \end{aligned}$$

inverter delay τ. .

□ Gate delay formula:

d = f + p.

 Effort delay f is related to gate's load. Parasitic delay p depends on gate's structure. Represents delay of gate driving no load Set by internal parasitic capacitance

Effort delay

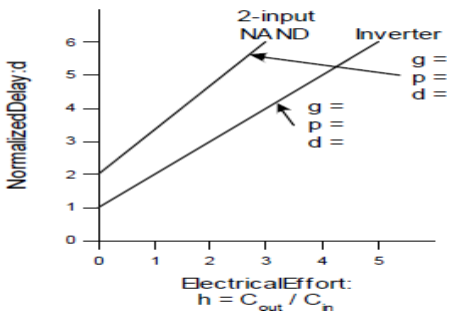
 Effort delay has two components: f = gh.

 Electrical effort h is determined by gate's load: h = Cout/Cin Sometimes called fanout

 Logical effort g is determined by gate's structure. Measures relative ability of gate to deliver current $g \equiv 1$ for inverter

Delay plots:

$$\begin{aligned} d &= f + p \\ &= gh + p \end{aligned}$$



Computing Logical Effort

□ Logical effort is the ratio of the input capacitance of a gate to the input capacitance of an inverter delivering the same output current. Measure from delay vs. fanout plots Or estimate by counting transistor widths

Sizing the transistors

$$D = \sum d_i = D_F + P$$

- Delay is smallest when each stage bears same effort
  - Optimal buffer chains are exponentially tapered:

    $$\hat{f} = g_i h_i = F^{\frac{1}{N}}$$

  - Thus minimum delay of N stage path is $D = NF^{\frac{1}{N}} + P$
- Determine W/L of each gate on path by working backward from the last gate: $\hat{f} = gh = g\frac{C_{out}}{C_{in}}$

$$\Rightarrow C_{in_i} = \frac{g_i C_{out_i}}{\hat{f}}$$

Power Estimation:

The past the major concerns of the VLSI designer were area performance cost and reliability power considerations were mostly of only secondary importance. In recent years however this has begun to change and increasingly power is being given comparable weight to area and speed. Several factors have contributed to this trend Perhaps the primary driving factor has been the remarkable success and growth of the class of personal computing devices portable desktops audio and videobased multimedia products_ and wireless communications systems _personal digital assistants and personal communicators_ which demand high_speed computation and complex functionality with In the past_ the major concerns of the VLSI designer were area_ performance cost and reliability_ power considerations were mostly of only secondary importance_ In recent years_ however_ this has begun to change and_ increasingly_ power is being given comparable weight to area and speed_ Several factors have contributed to this trend Perhaps the primary driving factor has been the remarkable success and growth of the class of personal computing devices _portable desktops_ audio_ and video_based multimedia products and wireless communications systems _personal digital assistants and personal communicators which demand high_speed computation and complex functionality with low power consumption_ There also exists a strong pressure for producers of high_end products to reduce their power consumption.

Software_Level Power Estimation

The first task in the estimation of power consumption of a digital system is to identify the typical application programs that will be executed on the system. A non_trivial application program consumes millions of machine cycles_ making it nearly impossible to perform power estimation using the complete program at_ say_ the RT_level_ Most of the reported results are based on power macro_modeling_ an estimation approach which is extensively used for behavioral and RTL level estimation see Sections and In the power cost of a CPU module is characterized by estimating the average capacitance that would switch when the given CPU module is activated_In the switching activities on _address_ instruction_ and data_ buses are used to estimate the power consumption of the microprocessor, based on actual current measurements of some processors_ Tiwari et al_ present the following instruction_level power model

$$Energy_p = \sum_i (BC_i N_i) + \sum_{i,j} (SC_{i,j} N_{i,j}) + \sum_k OC_k$$

where Energyp is the total energy dissipation of the program which is divided into three parts The first part is the summation of the base energy cost of each instruction, BCi is the base energy cost and Ni is the number of times instruction i is executed. The second part accounts for the circuit state SCi_j is the energy cost when instruction i is followed by j during the program execution_Finally_ the third part accounts for energy contribution OCk of other instruction effects such as stalls and cache misses during the program execution_ In Hsieh et al_ present a new approach_ called profile driven program synthesis_ to perform RT_level power estimation for high performance CPUs_ Instead of using a macro_modeling equation to model the energy dissipation of a microprocessor_ the authors use a synthesized program to exercise the microprocessor in such a way that the resulting instruction trace behaves _in terms of performance and power dissipation_ much the same as the original trace_ The new instruction trace is however much shorter than the original one_ and can hence be simulated on a RT_level description of the target microprocessor to provide the power dissipation results quickly_ Specifically_ this approach consists of the following steps_

- Perform architectural simulation of the target microprocessor under the instruction trace of typical application programs_
- Extract a characteristic pro_le_ including parameters such as the instruction mix_ Instruction data cache miss rates_ branch prediction miss rate_ pipeline stalls_ etc_ for the microprocessor.
- Use mixed integer linear programming and heuristic rules to gradually transform a generic program template into a fully functional program_
- Perform RT_level simulation of the target microprocessor under the instruction trace of the new synthesized program _

Notice that the performance of the architectural simulator in gate vectors second is roughly to orders of magnitude higher than that of a RT_level simulator. This approach has been applied to the Intel Pentium processor _which is a super_ scalar pipelined CPU with _KB _way set_associative data_ instruction and data caches_ branch prediction and dual instruction pipeline_ demonstrating _ to _ orders of magnitude reduction in the RT_level simulation time with negligible estimation error.

Behavioral_Level Power Estimation

Conversely from some of the RT_level methods that will be discussed in Section estimation techniques at the behavioral_level cannot rely on information about the gatelevel structure of the design components_ and hence_ must resort to abstract notions of physical capacitance and switching activity to predict power dissipation in the design_

Information_Theoretic Models

Information theoretic approaches for high_level power estimation depend on information theoretic measures of activity .for example_ entropy_ to obtain quick power estimates Entropy characterizes the randomness or uncertainty of a sequence of applied vectors and thus is intuitively related to switching activity_ that is_ if the signal switching is high_ it is likely that the bit sequence is random_ resulting in high entropy_ Suppose the sequence contains t distinct vectors and let pi denote the occurrence probability of any vector v in the sequence_ Obviously_

The entropy of the sequence is given

$$h = -\sum_{i=1}^{t} p_i \log p_i$$

where log x denotes the base logarithm of x_ The entropy achieves its maximum value of log t when pi log pi For an n_bit vector(t,n)his makes the computation of the exact entropy very expensive. Assuming that the individual bits in the vector are independent_ then we can write

$$h = -\sum_{i=1}^{n} (q_i \log q_i + (1 - q_i) \log (1 - q_i))$$

where qi denotes the signal probability of bit i in the vector sequence. Note that this equation is only an upperbound on the exact entropy, since the bits may be dependent. This upperbound expression is_ however_ the one that is used for power estimation purposes. Furthermore in it has been shown that_ under the temporal independence assumption_ the average switching activity of a bit is upper_bounded by one half of its entropy

The power dissipation in the circuit can be approximated as

$$Power = 0.5V^2 f C_{tot} E_{avg}$$

Where Ctot is the total capacitance of the logic module including gate and interconnect capacitances_ and Eavg is the average activity of each line in the circuit which is inturn approximated by one half of its average entropy havg. The average line entropy is computed by abstracting information obtained from a gate_level implementation. In it is assumed that the word_level entropy per logic level reduces quadratically from circuit inputs to circuit outputs_ whereas in it is assumed that the bit_level entropy from one logic level to next decreases in an exponential manner. Based on these assumptions two different computational models are obtained

In Marculescu et al_ derive a closed_form expression for the average line entropy for the case of a linear gate distribution(i.e.,)when the number of nodes scales linearly between the number of circuit inputs n and circuit outputs m. The expression for havg is given by

$$h_{avg} = \frac{2nh_{in}}{(n+m)\ln\frac{h_{in}}{h_{out}}} \left(1 - \frac{m}{n}\frac{h_{out}}{h_{in}} - \frac{(1-\frac{m}{n})(1-\frac{h_{out}}{h_{in}})}{\ln\frac{h_{in}}{h_{out}}}\right)$$

where hin and hout denote the average bit_level entropies of circuit inputs and outputs_respectively_ hin is extracted from the given input sequence_ whereas hout is calculated from a quick functional simulation of the circuit under the given input sequence or by empirical entropy propagation techniques for pre_characterized library modules. In Nemani and Najm propose the following expression for havg

$$h_{avg} = \frac{2}{3(n+m)}(H_{in} + H_{out})$$

where Hin and Hout denote the average sectional _word_level_ entropies of circuit inputs and outputs_ respectively_ The sectional entropy measures Hin and Hout may be obtained by monitoring the input and output signal values during a high_level simulation of the circuit_ In practice_ however_ they are approximated as the summation of individual bit_level entropies_ hin and hout. If the circuit structure is given_ the total module capacitance is calculated by traversing the circuit netlist and summing up the gate loadings_ Wire capacitances are estimated using statistical wire load models_ Otherwise_ Ctot is estimated by quick mapping for example_ mapping to__input universal gates_ or by information theoretic models that relate the gate complexity of a design to the di_erence of its input and output entropies. One such model proposed by Cheng and Agrawal in for example estimates

$$C_{tot} = \frac{m}{n} 2^n h_{out}$$

This estimate tends to be too pessimistic when n is large hence in Ferrandi et al_ present a new total capacitance estimate based on the number N of nodes i.e.,to multiplexors in the Ordered Binary Decision Diagrams OBDD representation of the logic circuit as follows

$$C_{tot} = \alpha \frac{m}{n} N h_{out} + \beta$$

The coefficients of the model are obtained empirically by doing linear regression analysis on the total capacitance values for a large number of synthesized circuits. Entropic models for the controller circuitry are proposed by Tyagi in where three entropic lower bounds on the average Hamming distance _bit changes_ with state set S and with T states_ are provided. The tightest lower bound derived in this paper for a sparse _nite state machine FSM i.e., tT log T where t is the total number of transitions with nonzero steady_state probability_ is the following

$$\sum_{s_i, s_j \in S} p_{i,j} H(s_i, s_j) \geq h(p_{i,j}) - 1.52 \log T - 2.16 + 0.5 \log(\log T)$$

where pi,j is the steady state transition probability from si to sj H(si,sj) is the Hamming distance between the two states_ and h(pi,j) is the entropy of the probability distribution pi,j . Notice that the lower bound is valid regardless of the state encoding used. In using a Markov chain model for the behavior of the states of the FSM_ the authors derive theoretical lower and upper bounds for the average Hamming distance on the state lines which are valid irrespective of the state encoding used in the final implementation. Experimental results obtained for the mcnc benchmark suite show that these bounds are tighter than the bounds reported.

Design Margin:

As semiconductor technology scales to the nanometer regime, the variation of process parameters is a critical problem in VLSI design. Thus the need for variation-aware timing analysis for the performance yield is increasing. However, the traditional worst-case corner-based approach gives pessimistic results, and makes meeting given designs specifications difficult. As an alternative to this approach, statistical analysis is proposed as a new and promising variation-aware analysis technique. However, statistical design flow cannot be applied easily to existing design flow, and not enough tools for statistical design exist. To overcome these problems, new design methodology based on traditional static timing analysis (STA) using a relaxed corner proposed nowadays. This paper investigates the effects of corner relaxation on overall circuit performance metrics (yield, power, area) at the gate/transistor levels. Experimental results indicate that if we design the circuit using relaxed corner, though the circuit yield is somewhat reduced, we can get some advantages in area and power aspects.

Reliability:

Yield and reliability are two of the cornerstones of a successful IC manufacturing technology along with product performance and cost. Many factors contribute to the achievement of high yield and reliability, and many of these also interact with product performance and cost. A fundamental understanding of failure mechanisms and yield limitations enables the up-front achievement of these technology goals through circuit and layout design, device design, materials choices, process optimization, and thermo-mechanical considerations. Failure isolation and analysis, defect analysis, low yield analysis, and materials analysis are critical methodologies for the improvement of yield and reliability. Coordination of people in many disciplines is needed in order to achieve high yield and reliability. Each needs to understand the impact of their choices and methods on the final product. Unfortunately, very little formal university training exists in these critical areas of IC reliability, yield, and failure analysis.

Reliability Fundamentals and Scaling Principles

- The Reliability Bathtub Curve, Its Origin and Implications
- Key Reliability Functions and Their Use in Reliability Analysis
- Defect Screening Techniques and Their Effectiveness
- Accelerated Testing and Estimation of Useful Operating Life
- Reliability Data Collection and Analysis in Integrated Circuits
- Past Technology Scaling Trends
- Forward Looking Projections with a Focus on Examining and Understanding of the Impact on VLSI Reliability
- Power Density Trends: Operating temperature, activation energies for dominant vlsi failure mechanisms, and reliability impact
- Reliability Strategies In Fables Environments

Reliability of the Interconnect System

- Physics and Statistics of Failure Mechanisms Associated with Interconnect Systems
- Electro-migration of Al and Cu Interconnects

- Mechanical Stress Driven Metal Voiding and Cracking
- Low k Materials as Interlayer Dielectrics and Their Impact on Electro-migration
- Thermo-mechanical Integrity of the Interconnect System
- Key Technology Parameters: Materials choices, structural and geometric effects
- Extreme Scaling Impact on Wear-out Time
- Technology Solutions: Alloys, metal barriers, and engineering of interfaces
- Improved Electro-migration Performance under Non-DC Currents and Short Lines
- Interconnect Reliability Strategies in Fabless Environments

Transistor Reliability: Dielectric Breakdown, Hot Carriers and Parametric Stability

- Physics, Statistics, and Scaling Impact on Failure Mechanisms
- Reliability Performance of Thin Conventional Oxides: Defects, wear-out failures
- Hot Carrier Performance and Parametric Stability of P- and N-channel Devices under DC and AC
- High k Gate Dielectrics and Novel Transistor Configurations
- Key Failure Mechanisms for Bipolar Transistors
- Transistor Reliability Strategies in Fabless Environments

CMOS Latch-up and ESD

- Physics, Scaling Impact, and Technology Dependence of CMOS Latch-up and Electrostatic Damage (ESD)
- Technology and Design Based Solutions, Device Performance, and Manufacturability Constraints
- Latch-up and ESD Assessment in Fabless Environments

Soft Errors, and Other Failure Mechanisms

- Physics, Scaling Impact, and Technology Dependence of Alpha Particle and Cosmic Ray Induced Soft Errors
- Technology Solutions, Performance, and Manufacturability

Scaling:

In scaling there are really two issues

• Devices

– Can we build smaller devices

– What will their performance be

• Wires

– Try to avoid the wet noodle effect

• There is concern about our ability to scale both of these Components

Limitations

Limitations to device scaling has been around since working in 3m nMOS, 22 years ago (actually bipolar)

• Worries were

– Short channel effect

– Punchthrough

• drain control of current rather than gate

– Hot electrons

– Parasitic resistances

• Now worries are a little different

– Oxide tunnel currents

– Punchthrough

– Parameter control

– Parasitic resistances

Transistor scaling:

People are building very short channel devices

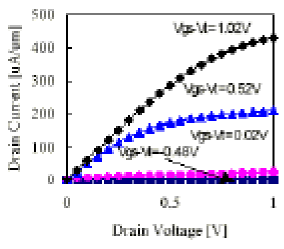
– Shown are I-V curves for 15nm L pMOS

– And a short channel nMOS

• The structure is strange

– FinFET

– But you can make them work



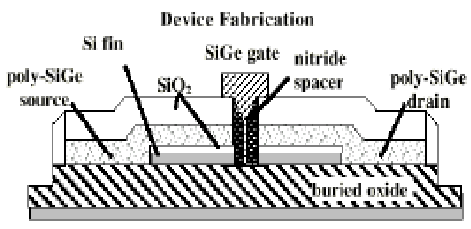
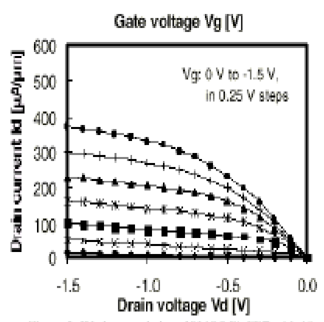


Figure 1: Schematic drawing of FinFET



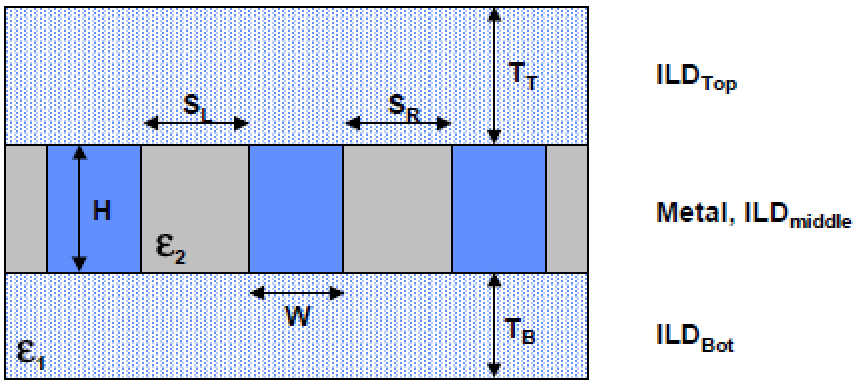
Wire scaling:

More uncertainty than transistor scaling

– Many options with complex trade-offs

• For each metal layer

– Need to set H, TT, TB, e1, e2, conductivity of the metal



SPICE Tutorial:

1. Introduction

Given below is a brief introduction to simulation using HSPICE and AWAVES/Cosmoscope in the UTD network. HSPICE is a device level circuit simulator from Synopsys. HSPICE takes a SPICE file as input and produces output describing the requested simulation of the circuit. The simulation output can be viewed with AWAVES (or) Cosmoscope from Synopsys. A short example is provided to illustrate the basic procedures involved in running HSPICE.

2. Setting up your account to access HSPICE

This section shows how to setup your environment for running HSPICE.

For users who have a working CAD setup, you may just want to check that the LM_LICENSE_FILE has the following values in the list of all the other licenses, ***/home/cad/flexlm/ti-license:/home/cad/flexlm/hspice.flx.*** If not, follow the procedures below: Instructions for both ***bash*** and ***tcsh/csh*** users is provided here:

bash users:

Add the following line to the .bash_profile

LM_LICENSE_FILE=$LM_LICENSE_FILE:/home/cad/flexlm/hspice.flx ; export LM_LICENSE_FILE

tcsh/csh users:

Add the following line to your .tcshrc

setenv LM_LICENSE_FILE ${LM_LICENSE_FILE}:/home/cad/flexlm/hspice.flx

To test if the above procedure has setup your environment successfully, invoke a new shell (this will ensure that the new environment variables are in place). Also you will need a HSPICE input file to test this (You can copy paste the HSPICE example given below to test this). The input Spice file is typically named with extension *.sp.

% hspice <your_input_file>.sp

The following message indicates **trouble** with invocation:

```
If the error is "hspice: command not found" make sure that the HSPICE
directory " /home/cad/synopsys/hspice/U-2003.09-SP1/sun58/" is included
in the $PATH variable.

Cannot execute /home/cad/synopsys/hspice/U-2003.09-SP1/sun58/hspice
or
lic: Using FLEXlm license file:
lic: /home/cad/flexlm/hspice.flx
lic: Unable to checkout hsptest
```

The above error may indicate that the license server maybe down, or the machine is not able to run HSPICE.

On the other hand if the procedure was successful, you will simply see a message indicating successful completion of simulation or errors in simulation, both of which indicate HSPICE has run your file.

3. Setting up the HSPICE input file

Consider a self loaded min geometry inverter circuit. The objective of the HSPICE input file below is to measure the t_{pLH} and t_{pHL} both graphically and otherwise. The following HSPICE file is stored in "inv.sp". The HSPICE input file is commented adequately about the different options used in it.

It will be beneficial to keep in mind the following differences between SPICE3 and HSPICE.

Property	**SPICE3**	**HSPICE**
Transistor dimensions	Default Scale is 1u.Hence depending on model, with or without "u".Eg. l=20	If units are not specified and no SCALE statement is present, the scale defaults to *meters*. Hence for HSPICE always specify units. Eg. l=20u
Input bit Pattern	PBIT or PWL	Only PWL format is supported. Howevere to convert a PBIT(Bit stream format) to a PWL form, you can use the script and help at the following page: http://www.utdallas.edu/~poras/courses/ee6325/lab/hspice/pbit2pwl.html
Output format	Print and Punch files produced only if requested as .punch/.print. SIMG reads the .pun file	SIMG does not read HSPICE output, only AWAVES or Cosmoscope can read HSPICE output. Also a graphical output produced only if .option post=1 is provided. The .print command is of no consequence to graphical output.
Line continuation	In order to specifying a continuing line '&' character is used at the end of the first line. Eg: Vin in gnd PWL& 0ns pvdd 1ns pvdd	In order to specifying a continuing line '+' character is used at the start of the second line. Eg: Vin in gnd PWL + 0ns pvdd 1ns pvdd

HSPICE Example File:

```
* Self loaded min geometry inverter, sample HSPICE file

* Include the model files

* Include the hspice model files for 0.18u technology.
.include  /home/cad/vlsi/models/hspice/cmos0.18um.model
*********************************************************************
* The subcircuit for the inverter

.subckt invert in out vdd gnd

.param length=0.2u
m01 out in vdd vdd pfet w='4*length' l='length'
m02 out in gnd gnd nfet w='1.5*length' l='length'

.ends
*********************************************************************

* The main inverter

X1 in out vdd gnd invert

* Four loads for the inverter

X2 out out1 vdd gnd invert
X3 out out2 vdd gnd invert
X4 out out3 vdd gnd invert
X5 out out4 vdd gnd invert

* PWL pattern for the input, represents a bit stream 1100101
* Slew=1ns, bit time=5ns
Vin in gnd PWL 0ns pvdd 1ns pvdd 5ns pvdd 6ns pvdd 7ns 0 10ns 0
+ 15ns 0 16ns pvdd 21ns pvdd 22ns 0 25ns 0 26ns pvdd

* Parametric definitions
```

```
.param pvdd=2.0v

* Power supplies
vvdd vdd 0 pvdd
vgnd gnd 0 0

* Control statements
.option post=1
.TR 0.05ns 30ns

.print TR V(in out)

* Measure statements help in calculating TPLH, TPHL etc, without
* opening the waveform viewer

.measure tran tplh trig v(in) val='0.5*pvdd' fall=1 targ v(out) val='0.5*pvdd' rise=1
.measure tran tphl trig v(in) val='0.5*pvdd' rise=1 targ v(out) val='0.5*pvdd' fall=1

.END
```

---

4. Running HSPICE simulations

The following commands can be used to simulate the above HSPICE file stored in **inv.sp** and store all the simulation results with file prefix as "**inv**"

```
% hspice inv.sp -o inv
```

This results in the creation of the following output files:
inv.ic -> Operating point node voltages (initial conditions)
inv.lis -> Output listing
inv.mt0 -> Transient analysis measurement results
inv.pa0 -> Subcircuit cross-listing
inv.st0 -> Output status
inv.tr0 -> Transient analysis results

5. Analyzing the outputs

In the above example, the output data can be analyzed both graphically as well as in text form.

Text outputs:

To view the results of the .measure computation, execute:

```
% cat inv.mt0
```

```
$DATA1 SOURCE='HSPICE' VERSION='2003.09-SP1'
.TITLE ' '
 tplh       tphl        temper     alter#
 3.416e-10  1.002e-09   25.0000    1.0000
```

As can be seen above, the values of propogation delay have been obtained even before the waveform analysis software has been opened.

Graphical outputs:

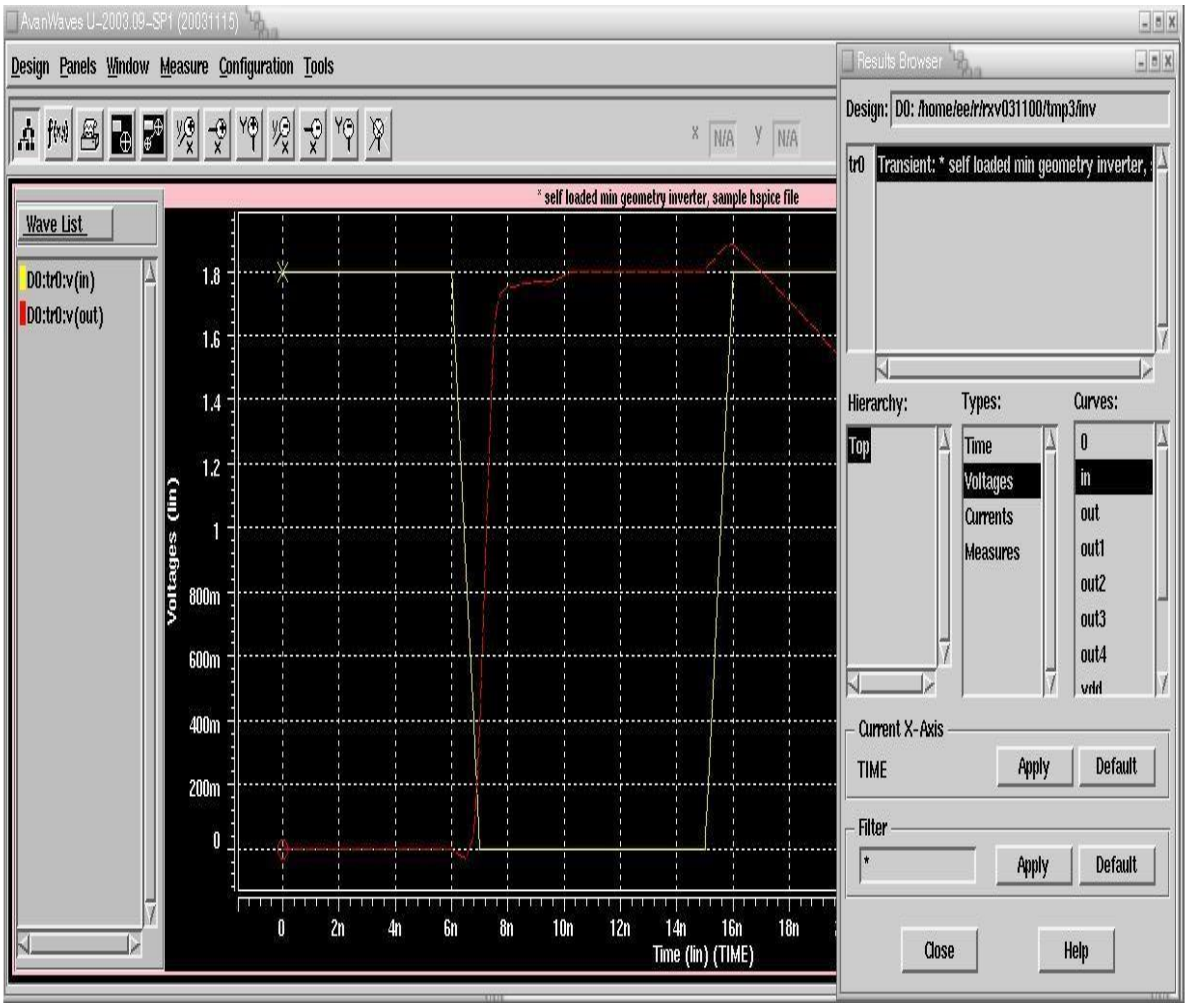
<u>I.</u> <u>Synopsys Awaves:</u>

To invoke AWAVES run the following command:

```
% awaves
```

If you get the error "awaves: command not found" make sure that the AWAVES directory "/home/cad/synopsys/hspice/U-2003.09-SP1/sun58/" is included in the $PATH variable.

Once invoked, open the design using the pull down menu options: Design->Open and select inv.sp and then highlight the tr0 (Transient response) item in the select box. You will also see the hierarchy of the netlist and the types of analysis and the individual signals in separate lists in the window. Select Hierarchy -> Top, Types -> Voltages and select the voltages you want to observe For eg. in and out by double clicking on the names. You will see the screen below for the stimulus provided in inv.sp.


AvanWaves U-2003.09-SP1 (20031115)
Design Panels Window Measure Configuration Tools
Wave List
D0:tr0:v(in)
D0:tr0:v(out)
* self loaded min geometry inverter, sample hspice file
Voltages (lin)
Time (lin) (TIME)
Results Browser
Design: D0: /home/ee/r/rxv031100/tmp3/inv
tr0 Transient: * self loaded min geometry inverter,
Hierarchy: Top
Types: Time Voltages Currents Measures
Curves: 0 in out out1 out2 out3 out4 vdd
Current X-Axis TIME Apply Default
Filter * Apply Default
Close Help

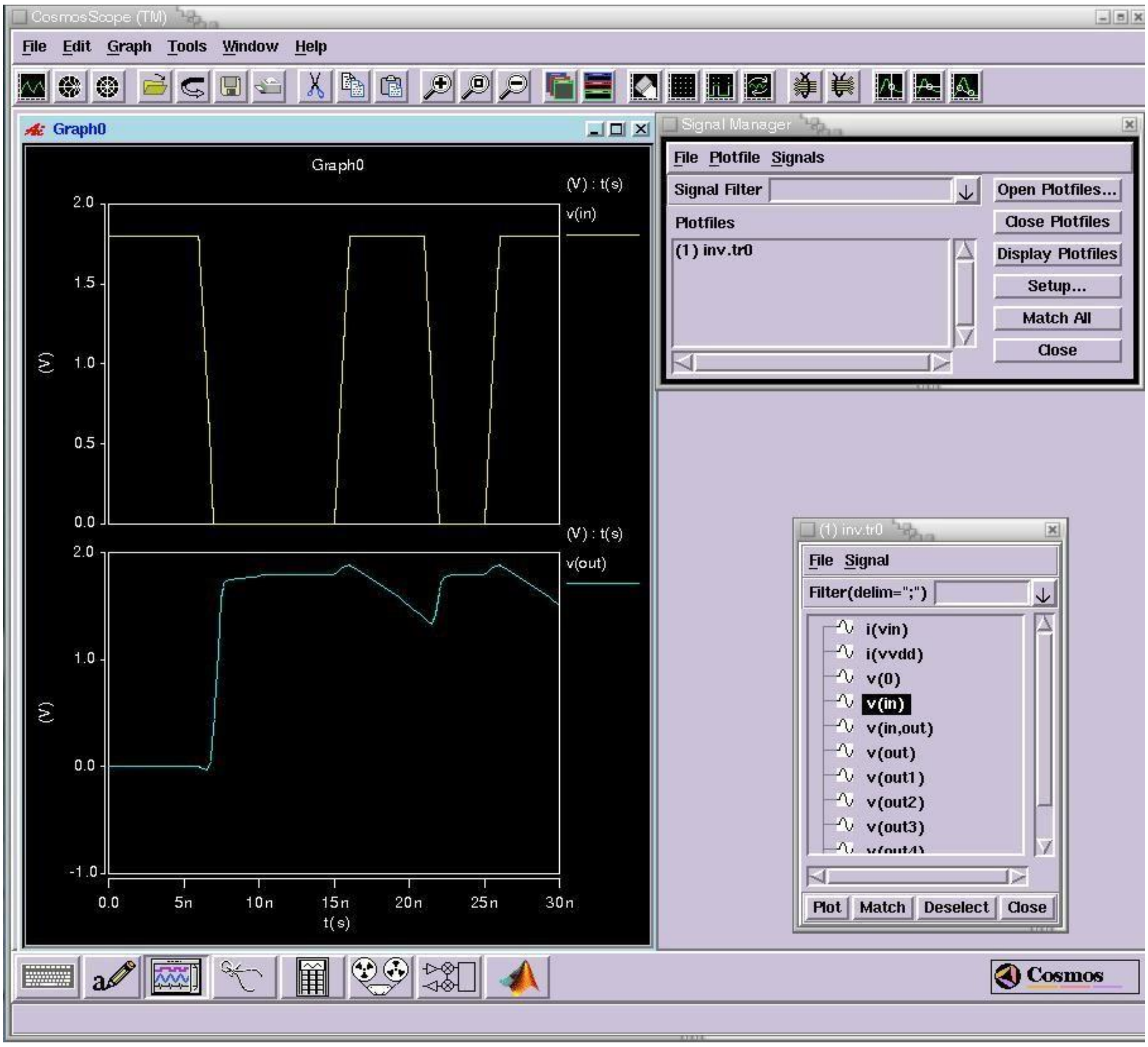
<u>II.</u> <u>Synopsys Cosmoscope:</u>

To invoke Cosmoscope run the following command:

%cscope

If you get the error "awaves: command not found" make sure that the AWAVES directory " /home/cad/synopsys/cosmo/ai_bin/" is included in the $PATH variable.

Once invoked, open the design using the pull down menu options: File -> Open -> Plotfiles and select file inv.tr0 in the working directory. A Signal Manager window and signal window opens. Select the necessary signals to be plotted by double-clicking them.For example v(in) and v(out) by double clicking on the signal names in the signal window. You will see the screen below for the stimulus provide



d in inv.sp.

Device models:

The motivation for this investigation stems from three main concerns:

1. The usual parameterization of device models for device and circuit simulation causes problems due to the interdependence of the parameters. It is not physically realistic to change any one parameter without determining the change in the process technology that would produce such a change in the parameter. Then all the other parameters which also depend on this change in the technology must be adjusted accordingly. In addition, it is quite difficult to determine the effect of a specific change in a new technology since the available parameters each depend on a number of technology parameters.

2. The predictive performance of present models is not good. It has usually been necessary to fabricate devices in any chosen technology, and extract parameters, and then fit the model to this specific technology by use of additional “adjustment” parameters. Of course, this procedure is

reasonable and useful once a technology has been chosen. However, it would be useful if the model could produce fairly accurate results if only the process specifications are used. Without such predictive accuracy it is difficult to make an initial choice of technology.

3. Most models have been developed for digital applications where devices operate above threshold and therefore are not strongly temperature sensitive. This causes problems for modeling analog circuits which use subthreshold operation. In particular, the temperature dependence of subthreshold behavior has not been fully explored. In many models some parameters which are temperature dependent have been assumed to be constant. Device and circuit models are all based on the physical properties of semiconductor materials, the dimensions of the devices, and on theoretical and empirical equations which are intended to model electrical behavior. The distinction between theoretical and empirical equations is often unclear. Most of the equations are substantially empirical. Of all the equations, one of the most fundamental, and problematic, is the equation for ni, the intrinsic carrier concentration of a semiconductor. The definition of ni derives from the thermodynamic equilibrium of electron and hole formation, based on the fact that the energy gap is a Gibbs energy. The equilibrium equation is

$$np = N_c N_v e^{-\frac{E_g}{kT}} \quad , \quad n_i = (np)^{1/2}$$

where n is the electron concentration, p is the hole concentration, Nc is the density of effective states in the conduction band, Nv is the density of effective states in the valence band, Eg is the band gap, k is Boltzmann’s constant and T is the absolute temperature. The carrier concentration is then given by It would appear to be a simple matter to substitute Si values for Nc, Nv, Eg, and the value of the constant kT to obtain an accurate value of ni. However, the theoretical and experimental knowledge required for accurate values of Nc and Nv is even now incomplete. In the early 1960’s, when Si-based circuits were beginning to be designed and fabricated very little was known about Nc and Nv, but estimates were required for practical use. This led to approximations based on work. The key approximation was that chosen by Grove in . This approximation is the still widely used

$$n_i = 1.45 \times 10^{10} \qquad at \quad T = 300K$$

Determination of Intrinsic Carrier Concentration (ni)

The empirical expressions for Eg from Bludau and for Nc and Nv from Sproul and Green provide the values needed for equation (2.1). Our derivation of the equation for ni follows Green. Therefore, the exciton binding energy term, using the value Exb = 14.7meV, is included in the band gap, Eg. In the past, this term has either been neglected, or in some cases a value of 10meV has been used. Green [9] and Sproul and Green are by far the best references for the history, theory, and experimental measurements leading to reliable values for ni. Thus, the equations are:

$$E_{g0} = 1.1700eV$$

$$E_g = E_{g0} + 1.059e^{-5T} - 6.05e^{-7T^2} \qquad (0 < T < 190K)$$

$$E_g = 1.1785 - 9.025e^{-5T} - 3.05e^{-7T^2} \qquad (150 < T <= 300K)$$

$$n_i = 1.640 \times 10^{15T^{1.706}} e^{-\frac{E_g}{2kT}}.$$

Circuit and device characterization:

The modeling procedure is introduced in this chapter, taking into consideration the requirements for a *good* MOSFET analog model, discussed in the previous chapter. We note here two main aspects of our modeling approach;

a. The model must describe *accurately* all the operating regions in order to be integrated in a circuit simulator.

b. The current, conductance, and transconductance must be continuous in all regions of operation.

Our main goal in this chapter is to determine the drain current for any combination of terminal voltages. The chapter is divided into two main parts. Throughout the first part, it is assumed that the channel is sufficiently *long* and *wide*, so that edge effects are confined to a negligible part of it. While in the second part we incorporate the short and narrow channel effects to the model. We also assume that the substrate is *uniformly* doped. The doping concentration will be assumed to be *p*-type and the modification to non uniform doping will be discussed later in this chapter.

Gradual Channel approximation (GCA)

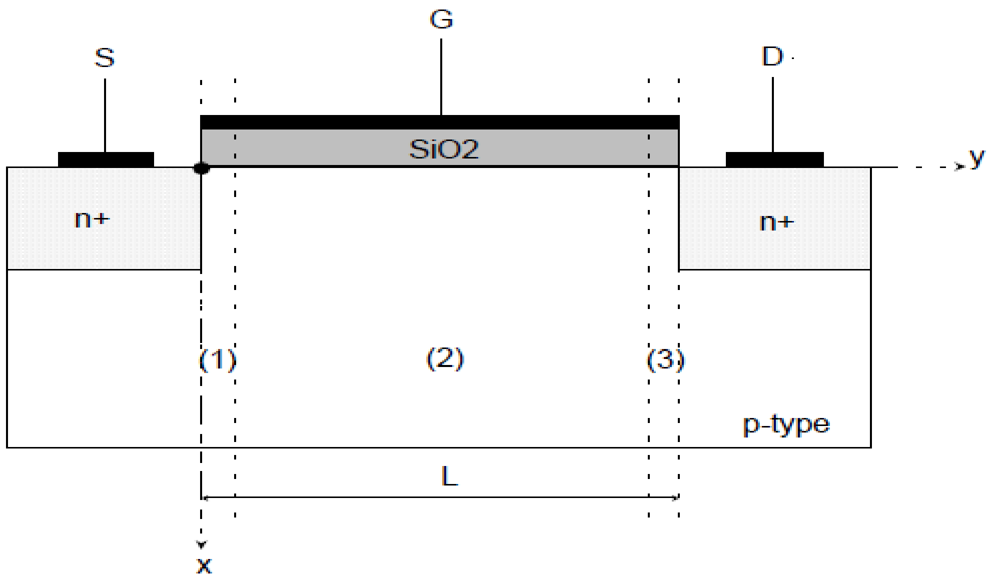


Figure 2.1 *The MOSFET structure.*

Analytical or semi-analytical modeling of MOSFET characteristics is usually based on the so-called *Gradual Channel Approximation* (GCA). In this approximation, we assume that the gradient of the electric field in the y direction, ∂F/∂y is much smaller than the gradient of the electric field in the x direction ,∂F/∂x. Which enable us to determine the inversion and depletion

charge densities under the gate in terms of a one-dimensional electrostatic problem for the direction perpendicular to the channel. By applying of the two dimensional Poisson's equation for the semiconductor, refer to Fig. 2.1 region (2),

$$\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} = \frac{\rho}{\varepsilon_s} \quad (2.2.1)$$

if we assume that the GCA is valid equation 2.2.1 may be approximated to the following one dimensional differential equation

$$\frac{\partial F_x}{\partial x} \approx \frac{\rho}{\varepsilon_s} \quad (2.2.2)$$

we approach the source and drain junctions, the GCA becomes invalid (Fig. 2.1 regions (1) and (3)) because of the increasing longitudinal field due to the pn junctions which make ∂F/∂y comparable or even larger than ∂F/∂x . However, for the long channel MOSFET's these transition regions can be neglected with respect to the total length of the device. In order to account for the effect of these regions, it is necessary to use two dimensional analysis requiring a numerical solution of 2.2.1.

Validity of the GCA

The validity of GCA can be checked by making rough estimates of the variation in the longitudinal and vertical field components. We will establish expressions that allow the GCA to be checked under strong inversion.

$$\frac{\partial F_x / \partial x}{\partial F_y / \partial y} = \left(\frac{\varepsilon_{ox} L}{\varepsilon_{si}}\right)^2 \frac{q(V_{GS} - V_T)^2}{kT.t_{ox}^2.V_{DS}} >> 1 \quad (2.2.3)$$

For a MOSFET at 300K with L = 1.0μm, tox = 30 nm, VGS-VT = 0.5, and VDS = 0.5 V, the left hand side of inequality 2.2.3 is ~ 2300, indicating that the the GCA is a very good approximation for such a MOSFET. This also implies that the GCA can be valid even in submicron MOSFETs, provided that VGSVT is not too small.

2.3 The long channel current model

The derivation of the dc drain current relationship recognizes that, in general, the current in the channel of a MOSFET can be caused by both drift and diffusion current. In an NMOSFET we may assume the following resonable approximation :

i- The drain current is mainly carried by electrons .

ii- The current flows almost in the y direction.

iii- No sources or sinks in the channel.

Note that in weak inversion the surface potential along the channel in long channel MOSFETs is almost constant. Thus ∂Fy/∂y is very small, implying that ∂Fy/∂y<<∂Fx/∂x. Thus in long channel MOSFET the GCA is valid both in strong and weak inversion regions . Which enable us to reach to the following general relationship of drain current.

General Drift-Diffusion current equation in MOSFET:

This is the drift-diffusion drain current of the form

$$I_D(y) = \mu_n W(-Q_i)\frac{d\phi_c}{dy} = \mu_n W(-Q_i(y))\frac{d\psi_s(y)}{dy} + \mu_n W\phi_t\frac{dQ_i(y)}{dy} \quad (2.3.1)$$

where µ*n* is the electron surface mobility in the channel, *W* is the channel width, *Qi* is the inversion charge density per unit area, φ*c* is the quasi Fermi potential (the difference between *Efn* at the surface of the semiconductor a d *Efp* in the bulk of the semiconductor), ψ*s* is the surface potential referenced to the bulk potential, and φ*t* is the thermal voltage (=*kT*/*q*). The first term is the *drift current component*, while the second term is the *diffusion current component*. In both components, µ*n* is the electrons' surface mobility being less than the mobility in the bulk due to surface scattering.

Voltage-Charge equation from the Transverse electric field:

In order to eliminate the electron charge density *Qi* term in the current charge equation, a second relationship is required that relates the electron charge density to the applied potentials. Using the relationship between voltage and charge appearing across the MOS capacitor we have

Cox (VG -φ ms -ψ s)= -(Qi+QB+Qox+Qit) (2.3.2)

where *VG* is the gate voltage referenced to the bulk potential, φ*ms* is the metal semiconductor work function difference, *QB* is the depletion (bulk impurity) charge density per unit area, *Qox* is the sum of the effective net oxide charge per unit area at the Si-SiO2 interface, and *Qit* is the interface trapped charge density per unit area. Different approximations have been introduced in order to express the different MOS charges (*QB, Qox, Qit)* in terms of the applied voltages, then

using eq. (2.3.2) to compute the inversion charge density *Qi*. The resulting charge is then used in eq. (2.3.1) to determine the drain current; Four main approaches then follow, after them we shall discuss the proposed approach recently developed in ICL1 and modified by this work.

Interconnect simulation models:

The classical long-channel Pao and Sah model

The Pao-Sah model [11,14], published in 1966, was the first advanced long channel MOSFET model to be developed. While it retained the GCA, it didn't invoke the depletion approximation and permitted carrier transport in the channel by both drift and diffusion current. The formulation of the drain current equation is therfore general, but as a result requires numerical integration in two dimensions, which limits its application in CAD tools.

Approximations:

i . Gradual Channel Approximation is used.

ii . Constant mobility is assumed.

iii . Uniform substrate doping is considered.

Advantages:

i . It is physically based.

ii . It gives a continuous representation of the device characteristics from weak to strong inversion even to the saturation mode of operation.

Disadvantages:

i. It requires excessive computational requriments since it requires numerical integration in two dimension, rendering it unsuitable to be used for circuit CAD.

The charge-sheet based models

The limited practical utility of the Pao-Sah model motivated a search for an approximate advanced analytical model, that is still accurate over a wide range of operating conditions. The charge sheet model, introduced separately by Bacarani and Brews in 1978, has become the most widely adopted long channel MOSFET model that is accurate over the entire range of inversion. In this model the inversion layer is supposed to be a charge sheet of infinitesimal thickness [11,15,16] (*charge sheet approximation*). The inversion charge density *Qi* can then be calculated in terms of the surface potential ψs. The drain current (2.3.1) is then expressed in terms of the surface potential at the source and drain boundaries of the channel.

Approximations:

i . Gradual Channel Approximation is used.

ii . The mobility is assumed to be proportional to the electric field and is constant with position along the channel.

iii . Uniform substrate doping is considered.

Advantages:

i . It is physically based.

ii . It gives a continuous representation of the device characteristics from weak to strong inversion even to the saturation mode of operation.

ii . The charge sheet approximation introduces negligibly small error, and it is more computationally efficient than the classical model.

Disadvantages:

i . The boundary surface potentials cannot be expressed explicitly in terms of the bias voltages applied to the device, but must be found by a numerical process.

ii . The model is not valid in depletion or accumulation.

Different approaches have been introduced to circumvent this disadvantage. In it is shown that accurate numerical solutions for these surface potentials can be obtained with negligible computation time penalty. In the surface potentials are computed using cubic splines functions. In the implicit equation including the surface potential is replaced by an approximate function. Although all of these approaches have given good results, they have neglected the effect of the *interface trap* charge which is important in determining the subthreshold characteristics of the device, namely the subthreshold swing (the gate voltage swing needed to reduce the current by one decade).

Bulk Charge Model

The Bulk Charge model also known as variable depletion charge model, was developed in 1964, describes the MOSFET drain current only in strong inversion but of course has less computational requirements.

Approximations :

i . Drift current component only is considered

ii . Constant surface potential is assumed

iii . Id considered zero below threshold

Advantages :

i . Less computational time than the charge sheet model

Disadvantages :

i . The subthreshold region not defined

Square law model

This model has great popularity, when a first estimate to device operation, or simulating a circuit with a large number of devices is required. This model is obtained from the bulk charge model, on the assumption that $VDS << 2\varphi f+VBS$.

i . Drift current component only is considered

ii . Constant surface potential is assumed

iii . Id considered zero below threshold

iv . $VDS << 2\varphi f+VBS$

Advantages :

i . Very small computational time than any other model

ii . Suitable for hand calculations

Disadvantages :

i . The subthreshold region is not defined ii . Overestimates the drain current in saturation region

Approximate models

There exists a large number of introduced approximate models. All of these models originate from Brews' charge sheet model, where approximations to the surface potentials in various operating regions of the device have been used. This leads to different current equations each valid only in a specific region. The resulting equations are then empirically joined using different mathematical conditions of continuity.

Advantages:

i . They have good accuracy in the desired region of operation.

ii . They are very efficient from the point of view of computational time.

Disadvantages:

i . The error increases in the transition regions between different modes of operations.

ii . They include many non-physical fitting parameters.

Modified charge sheet model

The last discussed MOSFET models, have a common illness, no interface charges are included which play a great role in subthreshold region. So a modified model to the charge sheet model, which include the effect of interface charges is carried out in ICL, and will be presented now. The derivation begins by rewriting equation (2.3.1) in the following form :

I D = *I D*1+ *I D*2 (2.3.6.1)

where *ID1* is due to the presence of drift:

$$I_{D1} = \frac{W}{L}\int_{\psi_{s0}}^{\psi_{sL}} \mu_n(-Q_i)d\psi_s$$

2.3.6.2)

and *ID2* is due to the presence of diffusion:

$$I_{D2} = \frac{W}{L}\phi_t\int_{Q_{i0}}^{Q_{iL}} \mu_n dQ_i$$

(2.3.6.3)

after mathematical manipulation and following the same approximations as charge sheet model we reach the following drain current equations:

$$I_{D1} = \frac{W}{L}\mu_{eff}C_{ox}[(V_G - V_{FBD})(\psi_{sL} - \psi_{so}) - \frac{1}{2}(\psi_{sL}{}^2 - \psi_{so}{}^2) - \frac{2}{3}\frac{\gamma}{\beta^{3/2}}((\beta\psi_{sL} - 1)^{3/2} - (\beta\psi_{so} - 1)^{3/2})]$$

(2.3.6.4)

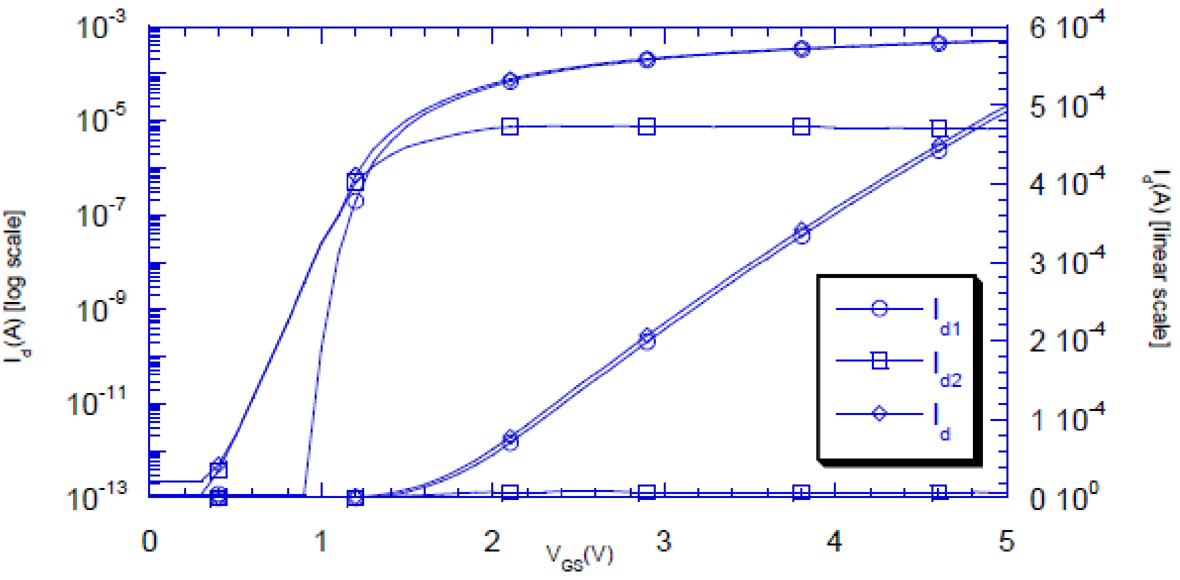
$$I_{D2} = \frac{W}{L}\frac{\mu_{eff}C_{ox}}{\beta}[(\psi_{sL}-\psi_{s0})+\frac{\gamma}{\sqrt{\beta}}(\sqrt{\beta\psi_{sL}-1}-\sqrt{\beta\psi_{s0}-1})$$
$$-\frac{1}{C_{ox}}(Q_{it}'(\psi_{sL},V_D)-Q_{it}'(\psi_{s0},V_S))] \qquad (2.3.6.5)$$

where ψ*s0* is the surface potential at the source end of the channel, ψ*sL* is the surface potential at the drain end of the channel, both are referred to the bulk. And their values are computed from the following two implicit equations.

$$C_{ox}(V_G-V_{FB}-\psi_{s0}) = \frac{\gamma C_{ox}}{\sqrt{\beta}}\sqrt{(\beta\psi_{s0}-1)+e^{\beta(\psi_{s0}-V_s-2\phi_f)}}$$
$$+\frac{q\,D_{it}}{\beta}[\beta(\psi_{s0}-V_s)+\ln(1+e^{-\beta(E_g/2-\phi_f+\psi_{s0}-V_s)})] \qquad (2.3.6.6)$$

$$C_{ox}(V_G-V_{FB}-\psi_{sL}) = \frac{\gamma C_{ox}}{\sqrt{\beta}}\sqrt{(\beta\psi_{sL}-1)+e^{\beta(\psi_{sL}-V_D-2\phi_f)}}$$
$$+\frac{q\,D_{it}}{\beta}[\beta(\psi_s-V_D)+\ln(1+e^{-\beta(E_g/2-\phi_f+\psi_{sL}-V_D)})]$$

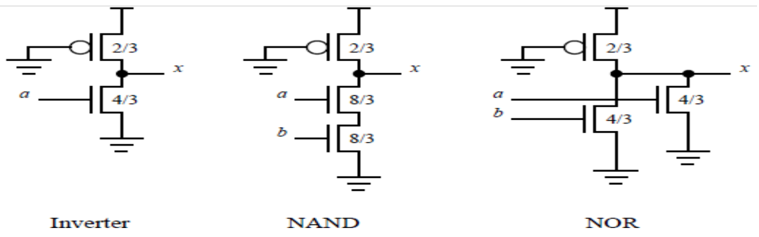
(2.3.6.7)



UNIT III COMBINATIONAL AND SEQUENTIAL CIRCUIT DESIGN

Circuit families and its comparison:

The method of logical effort does not apply to arbitrary transistor networks, but only to *logic gates.* A logic gate has one or more inputs and one output, subject to the following restrictions:
_ The gate of each transistor is connected to an input, a power supply, or the output; and _ Inputs are connected only to transistor gates. The first condition rules out multiple logic gates masquerading as one, and the second keeps inputs from being connected to transistor sources or drains, as in transmission gates without explicit drivers.



Pseudo-NMOS **circuits**

Static CMOS gates are slowed because an input must drive both NMOS and PMOS transistors. In any transition, either the pullup or pulldown network is activated, meaning the input capacitance of the inactive network loads the input. Moreover, PMOS transistors have poor mobility and must be sized larger to achieve comparable rising and falling delays, further increasing input capacitance. Pseudo-NMOS and dynamic gates offer improved speed by removing the PMOS transistors from loading the input. This section analyzes pseudo-NMOS gates, while section 10.2 explores dynamic logic. Pseudo-NMOS gates resemble static gates, but replace the slow PMOS pullup stack with a single grounded PMOS transistor which acts as a pullup resistor. The effective pullup resistance should be large enough that the NMOS transistors can pull the output to near ground, yet low enough to rapidly pull the output high. Figure 10.1 shows several pseudo-NMOS gates ratioed such that the pulldown transistors are about four times as strong as the pullup. The analysis presented in Section 9.1 applies to pseudo-NMOS designs. The logical effort follows from considering the output current and input capacitance compared to the reference inverter from Figure Sized as shown, the PMOS transistors produce 1/3 of the current of the reference inverter and the NMOS transistor stacks produce 4/3 of the current of the reference inverter. For falling transitions, the output current is the pulldown current minus the pullup current which is fighting the pulldown,For rising transitions, the output current

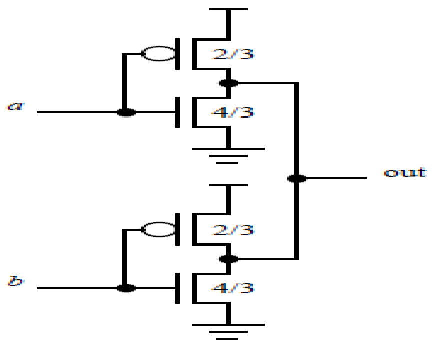
PSEUDO-NMOS CIRCUITS

Gate type	Logical Effort g		
	Rising	Falling	Average
2-NAND	8/3	8/9	16/9
3-NAND	4	4/3	8/3
4-NAND	16/3	16/9	32/9
n-NOR	4/3	4/9	8/9
n-mux	8/3	8/9	16/9

is just the pullup current, 1/3. The inverter and NOR gate have an input capacitance of 4/3. The falling logical effort is the input capacitance divided by that of an inverter with the same output current, or The rising logical effort is three times greater, because the current produced on a rising transition is only one third that of a falling transition. The average logical effort is g = (4=9+4=3)=2 = 8. This is independent of the number of inputs, explaining why pseudo-NMOS is a
way to build fast wide NOR gates. Table 10.1 shows the rising, falling, and average logical efforts of other pseudo-NMOS gates, assuming _ = 2 and a 4:1 pulldown to pullup strength ratio. Comparing this with Table 4.1 shows that pseudo-NMOS multiplexers are slightly better than CMOS multiplexers and that pseudo-NMOS NAND gates are worse than CMOS NAND gates. Since pseudo-NMOS logic consumes power even when not switching, it is best used for critical NOR functions where it shows greatest advantage. Similar analysis can be used to compute the logical effort of other logic technologies, such as classic NMOS and bipolar and GaAs. The logical efforts should be normalized so that an inverter in the particular technology has an average logical effort of 1.

10.1.1 Symmetric NOR **gates**

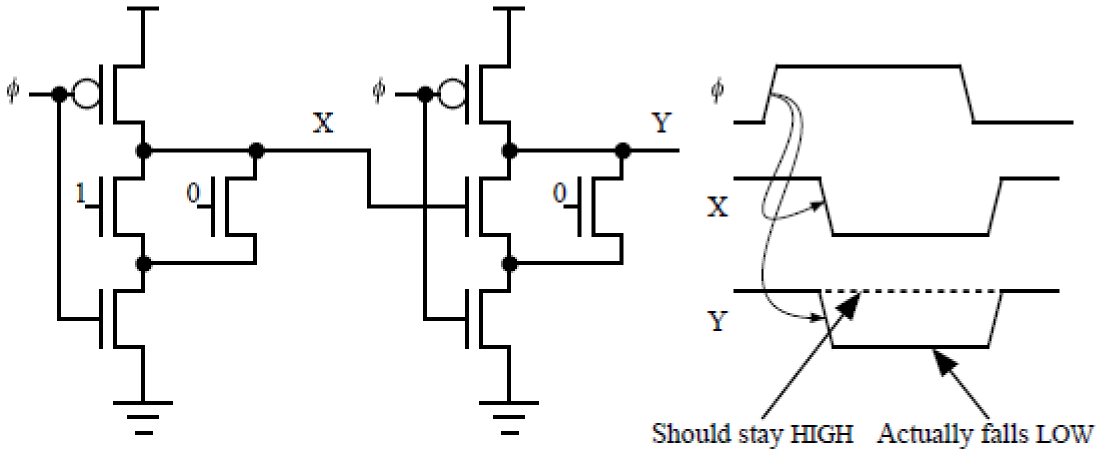
Johnson [4] proposed a novel structure for a 2-input NOR, shown in Figure 10.2. The gate consists of two inverters with shorted outputs, ratioed such that an inverter pulling down can overpower an inverter pulling up. This ratio is exactly the same as is used for pseudo-NMOS gates. The difference is that when the output should rise, both inverters pull up in parallel, providing more current than is available from a regular pseudo-NMOS pullup. The input capacitance of each input is 2. The worst-case pulldown current is



equal to that of a unit inverter, as we had found in the analysis of pseudo-NMOS NOR gates. The pullup current comes from two PMOS transistors in parallel and is thus 2=3 that of a unit inverter. Therefore, the logical effort is 2=3 for a falling output and 1 for a rising output. The average effort is g = 5=6, which is better than that of a pseudo-NMOS NOR and far superior to that of a static CMOS NOR! and even for NAND gates. Exercises 10-3 and 10-4 examine the design and logical effort of such structures.

Domino circuits

Pseudo-NMOS gates eliminate the bulky PMOS transistors loading the inputs, but pay the price of quiescent power dissipation and contention between the pullup and pulldown transistors. Dynamic gates offer even better logical effort and lower power consumption by using a clocked *precharge* transistor instead of a pullup that is always conducting. The dynamic gate is *precharged* HIGH then may *evaluate* LOW through an NMOS stack. Unfortunately, if one dynamic inverter directly drives another, a race can corrupt the result. When the clock rises, both outputs have been precharged HIGH.



The HIGH input to the first gate causes its output to fall, but the second gate's output also falls in response to its initial HIGH input. The circuit therefore produces an incorrect result because the second output will never rise during evaluation, as shown in Figure 10.3. *Domino* circuits solve this problem by using inverting static gates between dynamic gates so that the input to each dynamic gate is initially LOW. The falling dynamic output and rising static output ripple through a chain of gates like a chain of toppling dominos. In summary, domino logic runs 1:5 to 2 times faster than static CMOS logic [2] because dynamic gates present a much lower input capacitance for the same output current and have a lower switching threshold, and because the inverting static gate can be skewed to favor the critical monotonically rising evaluation edges. Figure shows some domino gates. Each *domino* gate consists of a dynamic gate followed by an inverting static gate1. The static gate is often but not always an inverter. Since the dynamic gate's output falls monotonically during evaluation, the static gate should be skewed high to favor its monotonically rising output. We have calculated the logical effort of high-skew gates in Table 9.2 and will compute the logical effort of dynamic gates in the next section. The logical effort of a domino gate is then the product of the logical effort of the dynamic gate and of the high-skew gate. Remember that a domino gate counts as two stages when choosing the best number of stages. A dynamic gate may be designed with or without a clocked evaluation transistor; the extra transistor slows the gate but eliminates any path between power and ground during precharge when the inputs are still high. Some dynamic gates include weak PMOS transistors called keepers so that the dynamic output will remain driven if the clock stops high. Domino designers face a number of questions when selecting a circuit topology. How many stages should be used? Should the static gates be inverters, or should they perform logic? How should precharge transistors and keepers be sized? What is the benefit of removing the clocked evaluation transistors? We will show that domino logic should be designed with a stage effort of 2–2:75, rather than 4 that we found for static logic. Therefore, paths tend to use more stages and it is rarely beneficial to perform logic with the inverting static gates.

Logic power logic design:

Is to Reduce dynamic power and static power in a circuit

– a:
– C:
– V_{DD}:
– f:

Reduce static power,Reduce dynamic power

– a: clock gating, sleep mode
– C: small transistors (esp. on clock), short wires
– V_{DD}:
– f:

Reduce static power, Reduce dynamic power

– a: clock gating, sleep mode
– C: small transistors (esp. on clock), short wires
– V_{DD}: lowest suitable voltage
– f: lowest suitable frequency

Reduce static power

– Selectively use ratioed circuits
– Selectively use low V_t devices
– Leakage reduction: stacked devices, body bias, low temperature.

Circuit design of latches and flip flops:

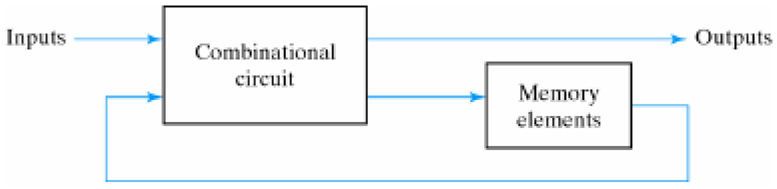
Another class of logic circuits are *sequential circuits*. These circuits are *two-valued* networks in which the outputs at any instant are dependent not only upon the inputs present at that instant but also upon the past history (sequence) of inputs.

Sequential circuits are classified into:

The block diagram of a sequential circuit is shown below:

- **Synchronous sequential circuits** – Their behaviour is determined by the values of the signals at only discrete instants of time.
- **Asynchronous sequential circuits** – Their behaviour is immediately affected by the input signal changes.

The basic logic element that provides memory in many sequential circuits is the *flip-flop*.

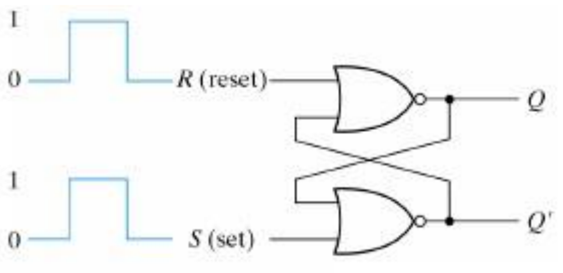


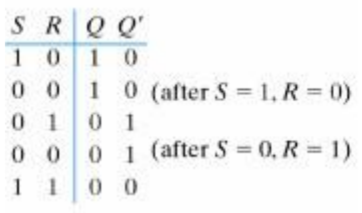
1. Latches

Latches form one class of flip-flops. This class is characterized by the fact that the timing of the output changes is not controlled. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use in synchronous sequential circuits.

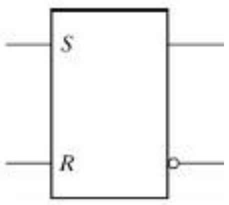
The *SR* Latch

It is a circuit with two cross-coupled NOR gates *or* two cross-coupled NAND gates. The one with NOR gates is shown below:

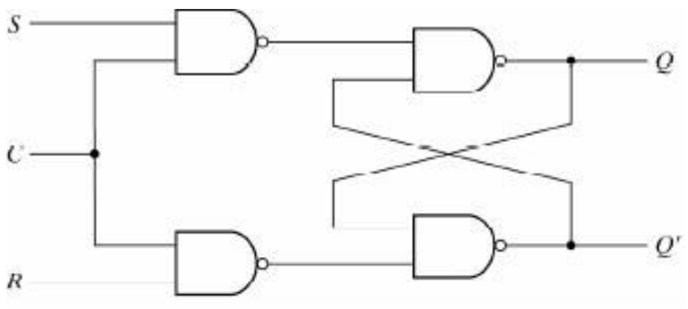




S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	



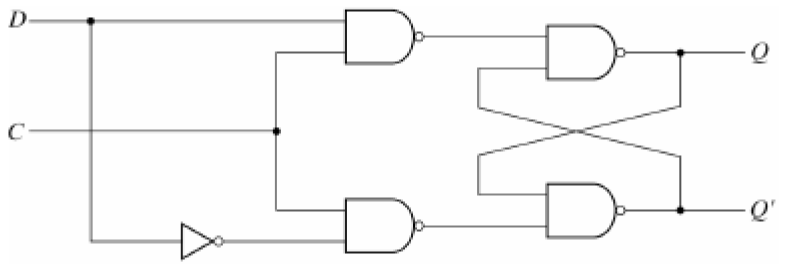
The condition that is undefined is when both inputs are equal to 0 at the same time. Comparing the NAND with the NOR latch note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an *S′*-*R′* latch. The operation of the *SR* latch can be modified to provide an additional control input that determines when the state of the latch can be changed. An *SR* latch with control input is shown below: The control input *C* acts as an enable signal form the other two inputs. An indeterminate condition occurs when all three inputs are equal to 1. This condition makes the circuit difficult to manage and is seldom used in practice. Nevertheless, it is an important circuit because other latches and flip-flops are realized from it.



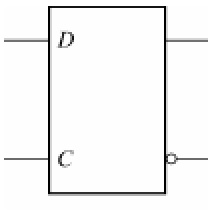
C	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; Reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

The Gated *D* Latch

One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done by the *D* latch:



C	D	Next state of Q
0	X	No change
1	0	$Q = 0$; Reset state
1	1	$Q = 1$; Set state

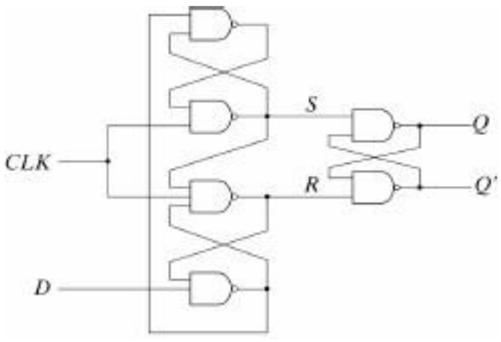


Flip-Flops

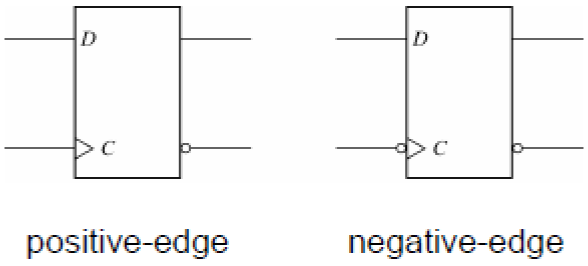
When latches are used for the memory elements in sequential circuits, a serious difficulty arises. Recall that latches have the property of immediate output responses (i.e., transparency). Because of this the output of a latch cannot be applied directly (or through logic) to the input of the same or another latch when all the latches are triggered by a common clock source. Flip-flops are used to overcome this difficulty.

Edge-Triggered *D* Flip-Flop

A *D* flip-flop may be realized with two *D* latches connected in a *master-slave* configuration: The circuit samples the *D* input and changes its *Q* output *only* at the negative-edge of the controlling clock signal (*CLK*). It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock (transition from 0 to 1). This happens in a flipflop that has an additional inverter between the *CLK* terminal and the junction between the other inverter and input *C* on the master latch. An efficient realization of a positive edge-triggered *D* flip-flop uses three *SR* latches:

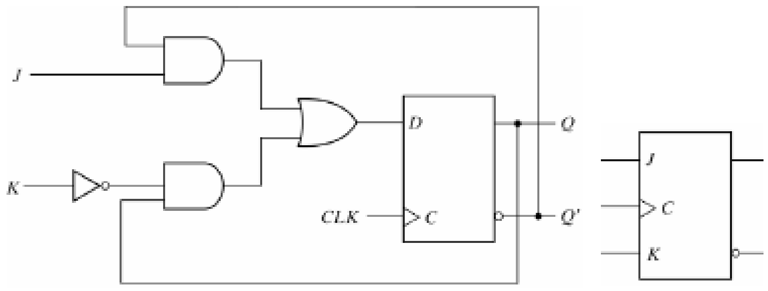


The graphic symbol for the edge-triggered *D* flipflop is:



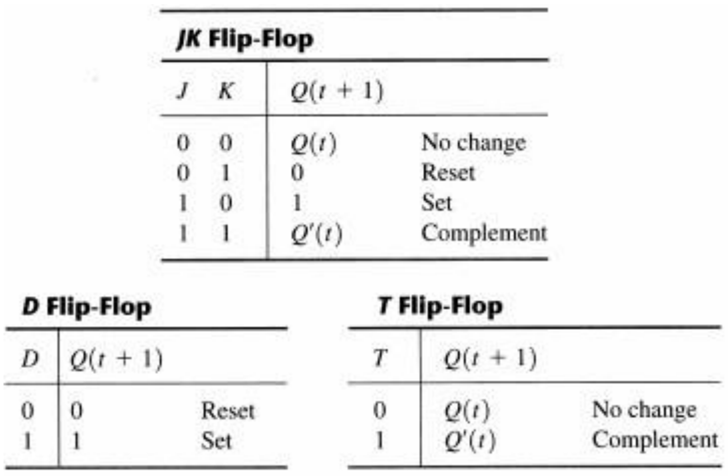
Other Flip-Flops

The most economical and efficient flip-flop in terms of transistor count and silicon area is the *D* flip-flop. Other types of flip-flops can be realized by using the *D* flip-flop and external logic. Two flipflops widely used in the design of digital systems are the *JK* and the *T* flip-flops. There are three operations that can be performed with a flip-flop: set it to 1, reset it to 0, complement its output. The *JK* flip-flop performs all three:



Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The flip-flops characteristic tables are:



***JK* Flip-Flop**

J	K	$Q(t+1)$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

***D* Flip-Flop**

D	$Q(t+1)$	
0	0	Reset
1	1	Set

***T* Flip-Flop**

T	$Q(t+1)$	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Q(*t*) refers to the present state prior to the application of a clock edge.

Q(*t* + 1) is the next state one clock period later. The clock edge input is not included in the characteristic tables, but is implied to occur between time t and *t* + 1.

Characteristic Equations

The logical properties of a flip-flop as described in its characteristic table can be expressed also algebraically with a characteristic equation. For the *D* flip-flop the characteristic equation is:

$Q(t+1) = D$

It states that the next state of the output will be equal to the value of input *D* in the present state.

The characteristic equation for the *JK* flip-flop is:

$$Q(t+1) = JQ' + K'Q$$

where *Q* is the value of the flip-flop output prior to the application of a clock edge. Finally, the characteristic equation for the *T* flipflop is:

$$Q(t+1) = T \oplus Q = TQ' + T'Q$$

Static sequencing element methodology:

Power has become critical metric and key differentiator in sub-65nm SOC designs, due to growing power density driven by technology scaling and chip integration. This tutorial provides overview of the low-power design methodologies and techniques in production SOC design perspective, emphasizing on the real design considerations and impact on chip success. We shall discuss pros and cons of the methods and techniques considering impacts on chip design schedule, yield, and overall power-performance target. We shall also provide design guidance and recommendations in various design steps and decision making points, based on our years of successful experience in production low-power SOC designs.

This tutorial is organized in two parts. In the first part, we shall overview power related challenges in sub-60nm SOC design and state-of-the-art techniques to reduce chip power. We shall give a holistic view from chip level to system and application levels. Practical industrial examples will be used to show how power savings can be achieved in modern SoC, processors and computer systems. In the second part, we shall describe production low-power design methodology and techniques particularly the power-gating and the voltage/frequency scaling which are the two advanced power reduction methods used effectively in sub-65nm production low-power designs. We shall explain when, where and how these methods and techniques are applied to a chip according to the design goals and time-to-market requirement. We shall also cover production low-power design methodology and flow with UPF power intent and unified design environment.

Sequencing dynamic circuits:

Simulation-based techniques for dynamic compaction of test sequences are proposed. The first technique uses a fault simulator to remove test vectors from the partially-specified test sequence generated by a deterministic test generator if the vectors are not needed to detect the target fault, considering that the circuit state may be known. The second technique uses genetic algorithms to fill the unspecified bits in the partially-specified test sequence in order to increase the number of faults detected by the sequence. Significant reductions in test set sizes were observed for all benchmark circuits studied. Fault coverages improved for many of the circuits, and execution times often dropped as well, since fewer faults had to be targeted by the computation-intensive deterministic test generator. 1 Introduction Deterministic test generators for single stuck-at faults in sequential circuits typically target individual faults, and once a test is generated, the test is fault simulated. CMOS inverters (Complementary NOSFET Inverters) are some of the most widely used and adaptable MOSFET inverters used in chip design. They operate with very little power loss and at relatively high speed. Furthermore, the CMOS inverter has good logic buffer characteristics, in that, its noise margins in both low and high states are large.

This short description of CMOS inverters gives a basic understanding of the how a CMOS inverter works. It will cover input/output characteristics, MOSFET states at different input voltages, and power losses due to electrical current. A CMOS inverter contains a PMOS and a NMOS transistor connected at the drain and gate terminals, a supply voltage VDD at the PMOS source terminal, and a ground connected at the NMOS source terminal, were VIN is connected to the gate terminals and VOUT is connected to the drain terminals.(See diagram). It is important to notice that the CMOS does not contain any resistors, which makes it more power efficient that a regular resistor-MOSFET inverter.As the voltage at the input of the CMOS device varies between 0 and 5 volts, the state of the NMOS and PMOS varies accordingly. If we model each transistor as a simple switch activated by VIN, the inverter's operations can be seen very easily.

UNIT IV CMOS TESTING

Need for testing:

Iddq testing is a method for testing CMOS integrated circuits for the presence of manufacturing faults. It relies on measuring the supply current (Idd) in the quiescent state (when the circuit is not switching and inputs are held at static values). The current consumed in the state is commonly called Iddq for Idd (quiescent) and hence the name. Iddq testing uses the principle that in a correctly operating quiescent CMOS digital circuit, there is no static current path between the power supply and ground, except for a small amount of leakage. Many common semiconductor manufacturing faults will cause the current to increase by orders of magnitude, which can be easily detected. This has the advantage of checking the chip for many possible faults with one measurement. Another advantage is that it may catch faults that are not found by conventional stuck-at fault test vectors. Iddq testing is somewhat more complex than just measuring the supply current. If a line is shorted to Vdd, for example, it will still draw no extra current if the gate driving the signal is attempting to set it to '1'. However, a different vector set that attempts to set the signal to 0 will show a large increase in quiescent current, signalling a bad part. Typical Iddq test vector sets may have 20 or so vectors. Note that Iddq test vectors require only controllability, and not observability. This is because the observability is through the shared power supply connection. Iddq testing has many advantages:

- It is a simple and direct test that can identify physical defects.
- The area and design time overhead are very low.
- Test generation is fast.
- Test application time is fast since the vector sets are small.
- It catches some defects that other tests, particularly *stuck-at* logic tests, do not.

Drawback: Compared to scan testing, Iddq testing is time consuming, and then more expensive, since is achieved by current measurements that take much more time than reading digital pins in mass production.

Testers and test programs:

As processes shrink (see Moore's law), the leakage current becomes much higher and less predictable. This makes it difficult to tell a low leakage part with a defect from a naturally high leakage part. Also, increasing circuit size means a single fault will have a lower percentage effect, making it harder for the test to detect. However, Iddq is so useful that designers are taking steps to keep it working. One particular technique that helps is power gating, where the entire power supply to each block can be switched off using a low leakage switch. This allows each block to be tested individually or in combination, which makes the tests much easier when compared to testing the whole chip.

Text fixtures:

In generic xUnit, a *test fixture* is all the things that must be in place in order to run a test and expect a particular outcome. Frequently fixtures are created by handling *setUp()* and *tearDown()* events of the unit testing framework. In *setUp()* one would create the expected state for the test, and in *tearDown()* it would clean up what had been set up.

Four phases of a test:

1. Set up -- Setting up the *test fixture*.
2. Exercise -- Interact with the *system under test*.
3. Verify -- Determine whether the expected outcome has been obtained.
4. Tear down -- Tear down the *test fixture* to return to the original state.

Use of fixtures

Some advantages of *fixtures* include separation of the test initialization (and destruction) from the testing, reusing a known state for more than one test, and special assumption by the testing framework that the fixture set up works.

Design for testability:

VLSI designers have a wide variety of CAD tools to choose from, each with their own strengths and weaknesses. The leading Electronic Design Automation (EDA) companies include Cadence, Synopsys, Magma, and Mentor Graphics. Tanner also offers commercial VLSI design tools. The leading free tools include Electric, Magic, and LASI. This set of laboratories uses the Cadence and Synopsys tools because they have the largest market share in industry, are capable of handling everything from simple class projects to state-of-the-art integrated circuits. The full set of tools is extremely expensive but the companies offer academic programs to make the tools available to universities at a much lower cost. The tools run on Linux and other flavors of Unix. Setting up and maintaining the tools involves a substantial effort. Once they are setup correctly, the basic tools are easy to use, as this tutorial demonstrates. Some companies use the Tanner tools because their list price is much lower and they are easy to use. However, their academic pricing is comparable with Cadence and Synopsys, giving little incentive for universities to adopt Tanner. The Electric VLSI Design System is an open-source chip design program developed by Electric presently does not read the design rules for state-of-the-art nanometer processes and poorly integrates with synthesis and place & route. Magic is a free Linux-based layout editor with a powerful but awkward interface that was once widely used in universities. The Layout System for Individuals, LASI, developed by David Boyce, is freely available and runs on Windows. It was last updated in 1999. There are two general strategies for chip design. *Custom design* involves specifying how every transistor is connected and physically arranged on the chip. *Synthesized design* involves describing the function of a digital chip in a hardware description language such as Verilog or VHDL, then using a computer-aided design tool to automatically generate a set of gates that perform this function, place the gates on the chip, and route the wires to connect the connect the gates. The majority of commercial designs are synthesized today because synthesis takes less engineering time. However, custom design gives more insight into how chips are built and into what to do when things go wrong. Custom design also offers higher performance, lower power, and smaller chip size. The first two labs emphasize the fundamentals of custom design, while the next two use logic synthesis and automatic placement to save time.

II. Tool Setup

These labs assume that you have the Cadence and Synopsys tools installed.
The tools generate a bunch of random files. It's best to keep them in one place. In your
home directory, create some directories by typing:

```
mkdir IC_CAD
mkdir IC_CAD/cadence
```

III. Getting Started

Before you start the Cadence tools, change into the cadence directory:

```
cd ~/IC_CAD/cadence
```

Each of our tools has a startup script that sets the appropriate paths to the tools and invokes them. Start Cadence with the NCSU extensions by running cad-ncsu & A window labeled icfb will open up. This is the Integrated Circuit Front and Back End (e.g. schematic and layout) software, part of Cadence's Design Framework interface. A "What's New" and a Library Manager window may open up too. Scroll through the icfb window and look at the messages displayed as the tool loads up. Get in the habit of watching for the messages and recognizing any that are out of the ordinary. This is very helpful when you encounter problems.

All of your designs are stored in a *library*. If the Library Browser doesn't open, choose Tools • Library Manager. You'll use the Library Manager to manipulate your libraries. Don't try to move libraries around or rename them directly in Linux; there is some funny behavior and you are likely to break them. Familiarize yourself with the Library Manager. Your cds.lib file includes many libraries from the NCUS CDK supporting the different MOSIS processes. It also includes libraries from the University of Utah. The File menu allows you to create new libraries and cells within a library, while the Edit menu allows you to copy, rename, delete, and change the access permissions.

Create a library by invoking File • New • Library… in the Library Manager. Name the library lab1_xx, where xx are your initials. Leave the path blank and it will be put in your current working directory (~/IC_CAD/cadence). Choose the "Attach to existing tech library" and accept the default, UofU AMI 0.60u C5N (3M, 2P, high-res). This is a technology file for the American Microsystesm (now Orbit Semiconductor) 0.6 μm process, containing design rules for layout.

IV. Schematic Entry

Our first step is to create a schematic for a 2-input NAND gate. Each gate or larger component is called a *cell*. Cells have multiple views. The schematic view for a cell built with CMOS transistors will be called cmos_sch. Later, you will build a view called layout specifying how the cell is physically manufactured. In the Library Manager, choose File • New • Cell View… In your lab1_xx library, enter a cell name of nand2 and a view name of cmos_sch. The tool should be Composer-Schematic. You may get a window asking you to confirm that cmos_sch should be associated with this tool. The schematic editor window will open. Your goal is to draw a gate like the one shown in Figure 1. We are working in a 0.6 μm process with $\lambda = 0.3$ μm. Unfortunately, the University of Utah technology file is configured on a half-lambda grid, so grid units are 0.15 μm. Take care that everything you do is an integer multiple of λ so you don't come to grief later on. Our NAND gate will use 12 λ (3.6 μm) nMOS and pMOS transistors.

Choose Add • Instance to open a Component Browser window. (The menu lists the keyboard shortcut for each command, such as *i* for add instance. You'll want to learn the shortcuts you use most often.) Choose UofU_Analog_Parts for the library, then select nmos. The Add Instance dialog will open. Set the Width to 3.6u (u indicates microns). Click in the schematic editor window to drop the transistor. You can click a second time to place another transistor. Return to the Component Browser window and choose pmos. Drop two pMOS transistors. Then return to the browser and get a gnd and a vdd symbol. When you are in a mode in the editor, you can press ctrl-c or Esc to get out of it. Other extremely useful commands include Edit • Move, Edit • Copy, Edit • Undo, and Edit • Delete. Edit • Properties • Object… is also useful to change things like transistor sizes or wire names. Move the elements around until they are in attractive locations. I like to keep series transistors onew grid unit apart and place pMOS transistors two grid units above the nMOS. Look at the bottom of the schematic editor window to see what mode you are in. Next, use Add • Pin… to create some pins. In the Add Pin dialog, enter a and b. Make sure the direction is "input." The tools are case-sensitive, so use lower case everywhere. Place the pins, being sure that a is the bottom one. Although pin order doesn’t matter logically, it does matter physically and electrically, so you will get errors if you reverse the order. Then place an output pin y. Now, wire the elements together. Choose Add • Wire (narrow). Click on each component and draw a wire to where it should connect. It is a good idea to make sure every net (wire) in a design has a name. Otherwise, you’ll have a tough time tracking down a problem later on one of the unnamed nets. Every net in your schematic is connected to a named pin or to power or ground except the net between the two series nMOS transistors. Choose Add • Wire name… Enter mid or something like that as the name, and click on the wire to name it. Choose Design • Check and Save to save your schematic. You’ll probably get one warning about a “solder dot on crossover” at the 4-way junction on the output node. This is annoying because such 4-way junctions are normal and common. Choose Check • Rules Setup… and click on the Physical tab in the dialog. Change Solder On CrossOver from “warning” to “ignored” and close the dialog. Then Check and Save again and the warning should be gone. If you have any other warnings, fix them. A common mistake is wires that look like they might touch but don’t actually connect. Delete the wire and redraw it. Poke around the menus and familiarize yourself with the other capabilities of the schematic editor.

V. Logic Verification

Cells are commonly described at three levels of abstraction. The register-transfer level (RTL) description is a Verilog or VHDL file specifying the behavior of the cell in terms of registers and combinational logic. It often serves as the specification of what the chip should do. The schematic illustrates how the cell is composed from transistors or other cells. The layout shows how the transistors or cells are physically arranged. Logic verification involves proving that the cells perform the correct function. One way to do this is to simulate the cell and apply a set of 1’s and 0’s called test vectors to the inputs, then check that the outputs match expectation. Typically, logic verification is done first on the RTL to check that the specification is correct. A testbench written in Verilog or VHDL automates the process of applying and checking all of the vectors. The same test vectors are then applied to the schematic to check that the schematic matches the RTL. Later, we will use a layout-versus schematic (LVS) tool to check that the layout matches the schematic (and, by inference, the RTL). You will begin by simulating an RTL description of the NAND gate to become familiar with reading RTL and understanding a testbench. In this tutorial, the RTL and testbench are written in System Verilog, which is a 2005 update to the popular Verilog hardware description language.

There are many Verilog simulators on the market, including NC-Verilog from Cadence, VCS from Synopsys, and ModelSim from Mentor Graphics. This tutorial describes how to use NC Verilog because it integrates gracefully with the other Cadence tools. NCVerilog compiles your Verilog into an executable program and runs it directly, making it much faster than the older interpreted simulators. Make a new directory for simulation (e.g. nand2sim). Copy nand2.sv, nand2.tv, and testfixture.verilog from the course directory into your new directory.

```
mkdir nand2sim
cd nand2sim
cp /courses/e158/10/nand2.sv .
cp /courses/e158/10/nand2.tv .
cp /courses/e158/10/nand2.testfixture testfixture.verilog
```

nand2.sv is the SystemVerilog RTL file, which includes a behavioral description of a nand2 module and a simple self-checking testbench that includes
testfixture.verilog. testfixture.verilog reads in testvectors from
nand2.tv and applies them to pins of the nand2 module. After each cycle it compares the output of the nand2 module to the expected output, and prints an error if they do not match. Look over each of these files and understand how they work. First, you will simulate the nand2 RTL to practice the process and ensure that the testbench works. Later, you will replace the behavioral nand2 module with one generated from your Electric schematic and will resimulate to check that your schematic performs the correct function. At the command line, type sim-nc nand2.sv to invoke the simulator. You should see some messages ending with

```
ncsim> run
Completed 4 tests with 0 errors.
Simulation stopped via $stop(1) at time 81 NS + 0
```

You'll be left at the ncsim command prompt. Type quit to finish the simulation. If the simulation hadn't run correctly, it would be helpful to be able to view the results. NC-Verilog has a graphical user interface called SimVision. The GUI takes a few seconds to load, so you may prefer to run it only when you need to debug. To rerun the simulation with the GUI, type sim-ncg nand2.sv A Console and Design Browser window will pop up. In the browser, click on the + symbol beside the testbench to expand, then click on dut. The three signals, a, b, and y,
will appear in the pane to the right. Select all three, then right-click and choose Send to Waveform Window. In the Waveform Window, choose Simulation • Run. You'll see the waveforms of your simulation; inspect them to ensure they are correct. The 0 errors message should also appear in the console. If you needed to change something in your code or testbench or test vectors, or wanted to add other signals, do so and then Simulation • Reinvoke Simulator to recompile everything and bring you back to the start. Then choose Run again. Make a habit of looking at the messages in the console window and learning what is normal. Warnings and errors should be taken seriously; they usually indicate real problems that will catch you later if you don't fix them.

VI. Schematic Simulation

Next, you will verify your schematic by generating a Verilog deck and pasting it into the RTL Verilog file. While viewing your schematic, click on Tools • Simulation • NCVerilog to open a window for the Verilog environment. Note the run directory (e.g. nand2_run1), and press the button in the upper left to initialize the design. Then press the next button to generate a netlist. Look in the icfb

window for errors and correct them if necessary. You should see that the pmos, nmos, and nand2 cells were all netlisted. In your Linux terminal window, cd into the directory that was created. You'll find quite a few files. The most important are verilog.inpfiles,
testfixture.template, and testfixture.verilog. Each cell is netlisted into a different directory under ihnl. verilog.inpfiles states where they are. Take a look at the netlist and other files. testfixture.template is the top level module that instantiates the device under test and invokes the testfixture.verilog. Copy your from your nand2sim directory to your nand2_run1 directory using a command such as

```
cp ../nand2sim/testfixture.verilog .
cp ../nand2sim/nand2.tv .
```

Back in the Virtuoso Verilog Environment window, you may wish to choose Setup • Record Signals. Click on the "All" button to record signals at all levels of the hierarchy. (This isn't important for the nand with only one level of hierarchy, but will be helpful later.) Then choose Setup • Simulation. Change the Simulation Log File to indicate simout.tmp –sv. This will print the results in simout.tmp. The –sv flag indicates that the simulator should accept SystemVerilog syntax used in the testfixture.verilog. Set the Simulator mode to "Batch" and click on the Simulate button. You should get a message that the batch simulation succeeded. This doesn't mean that it is correct, merely that it run. In the terminal window, view the simout.tmp file. It will give some statistics about the compilation, then should indicate that the 4 tests were completed with 0 errors. If the simulation fails, the simout.tmp file will have clues about the problems. Change the simulator mode to Interactive to rerun with the GUI. Be patient; the GUI takes several seconds to start and gives no sign of life until then. Add the waveforms again and run the simulation. You may need to zoom to fit all the waves. For some reason, SimVision doesn't print the $display message about the simulation succeeding with no errors. You will have to read the simout.tmp file at the command line to verify that the test vectors passed. If you find any logic errors, correct the schematic and resimulate.

Silicon Debug

The rapid pace of innovation has created powerful SOC solutions at consumer prices. This has created a highly competitive market place where billions of dollars can be won by the right design delivered at the right time. These new designs are produced on processes that challenge the fundamental laws of physics and are highly sensitive to equipment variation. The industry now produces new designs in a complex world where process and design interaction have created new complex failures that stand in the way of billion-dollar opportunities. These interactions lead to new types of defects such as blocked chains, which create noise in the debug/diagnosis process. They also lead to new types of design issues such as delay defects in combinational and sequential logic.

The challenge is made even greater by the growing complexity in device structure and design techniques. Multiple design organizations use multiple IP blocks and multiple libraries that need to work together throughout the process window, often across multiple fabs. These new challenges come at a time when product lifetimes are shrinking, leading to pressure to reduce time for debug and characterization activities. These problems are seen for the first time at first silicon. Test the first chips back from fabrication If you are lucky, they work the first time If not . . . Logic bugs vs. electrical failures Most chip failures are logic bugs from inadequate simulation or verification 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 Silicon debug (or "bringup") is primarily a Non-Recurring Engineering (NRE) cost (like design) Contrast this with manufacturing test which has to be applied to every part shipped.

Manufacturing test:

A speck of dust on a wafer is sufficient to kill chip Yield of any chip is < 100% Must test chips after manufacturing before delivery to customers to only ship good parts Manufacturing testers are very expensive Minimize time on tester Careful selection of test vectors. A test for a defect will produce an output response which is different from the output when there is no defect Test quality is high if the set of tests will detect a very high fraction of possible defects Defect level is the percentage of bad parts shipped to customers Yield is the percentage of defect-free chips manufactured
Fault Models

Fault models:

Numerous possible physical failures (what we are testing for) Can reduce the number of failure types by considering the effects of physical failures on the logic functional blocks: called a Assume that defects will cause the circuit to behave as if lines were "stuck" at logic 0 or 1 Most commercial tools for test are based on the "stuck-at" model Other fault models "Stuck open" model for charge retained on a CMOS node Recent use of the "transition" fault model in an attempt to deal with delays "Path delay" fault model would be better for small delay defects, but the large number of possible paths is an impediment to the use of this fault model.

Test for faults in a circuit:

Approach to generating tests for defects is to map defects to (higher level) faults: develop fault model, then generate tests for the faults Typical: gate-level "stuck-at" fault model As technology shrinks, other faults: bridging faults, delay faults, crosstalk faults, etc. An interesting point: what is important is how well the tests generated (based on the fault model) will detect realistic defects The accuracy of the fault model is secondary

Observability and controllability:

Observability: ease of observing a value on a node by monitoring external output pins of the chip Controllability: ease of forcing a node to 0 or 1 by driving input pins of the chip Combinational logic is usually easier to observe and control Still, NP-complete problem Finite state machines can be very difficult, requiring many cycles to enter desired state Especially if state transition diagram is not known to the test engineer, or is too large

Fault simulation:

Identify faults detected by a sequence of tests Provide a numerical value of coverage (ratio of detected faults to total faults) Correlation between high fault coverage and low defect level
Faults considered Generally, gate level “stuck-at” faults Can also evaluate coverage of switch level faults Can include timing and dynamic effects of failures Although fault simulation takes polynomial time in the number of gates, it can still be prohibitive for large designs. Static timing analysis (Primetime, for example) only finds structural long paths
False Path problem:
In order to allow a signal to go through the path, Required Side Inputs: C = 1, A = 1, E = 1 Conflict due to C = 1 and E = 1 Can use modified test generation algorithms to identify longest true paths in a circuit CRITIC from UT Primetime+Tetramax from Synopsys.

Boundary scan:

Boundary scan is a method for testing interconnects (wire lines) on printed circuit boards or sub-blocks inside an integrated circuit. Boundary scan is also widely used as a debugging method to watch integrated circuit pin states, measure voltage, or analyze sub-blocks inside an integrated circuit.

Testing

The boundary scan architecture provides a means to test interconnects and clusters of logic, memories etc. without using physical test probes. It adds one or more so called 'test cells' connected to each pin of the device that can selectively override the functionality of that pin. These cells can be programmed via the JTAG scan chain to drive a signal onto a pin and across an individual trace on the board. The cell at the destination of the board trace can then be programmed to read the value at the pin, verifying the board trace properly connects the two pins. If the trace is shorted to another signal or if the trace has been cut, the correct signal value will not show up at the destination pin, and the board will be observed to have a fault.

On-Chip Infrastructure

To provide the boundary scan capability, IC vendors add additional logic to each of their devices, including scan cells for each of the external traces. These cells are then connected together to form the external boundary scan shift register (BSR), and combined with JTAG TAP (Test Access Port) controller support comprising four (or sometimes more) additional pins plus control circuitry.

Some TAP controllers support scan chains between on-chip logical design blocks, with JTAG instructions which operate on those internal scan chains instead of the BSR. This can allow those integrated components to be tested as if they were separate chips on a board. On-chip debugging solutions are heavy users of such internal scan chains.

These designs are part of most Verilog or VHDL libraries. Overhead for this additional logic is minimal, and generally is well worth the price to enable efficient testing at the board level.

For normal operation, the added boundary scan latch cells are set so that they have no effect on the circuit, and are therefore effectively invisible. However, when the circuit is set into a test mode, the latches enable a data stream to be shifted from one latch into the next. Once a complete data word has been shifted into the circuit under test, it can be latched into place so it drives external signals. Shifting the word also generally returns the input values from the signals configured as inputs.

Test Mechanism

As the cells can be used to force data into the board, they can set up test conditions. The relevant states can then be fed back into the test system by clocking the data word back so that it can be analyzed.

By adopting this technique, it is possible for a test system to gain test access to a board. As most of today’s boards are very densely populated with components and tracks, it is very difficult for test systems to physically access the relevant areas of the board to enable them to test the board. Boundary scan makes access possible without always needing physical probes.

In modern chip and board design, Design For Test is a significant issue, and one common design artifact is a set of boundary scan test vectors, possibly delivered in Serial Vector Format (SVF) or a similar interchange format.

JTAG Test Operations

Devices communicate to the world via a set of input and output pins. By themselves, these pins provide limited visibility into the workings of the device. However, devices that support boundary scan contain a shift-register cell for each signal pin of the device. These registers are connected in a dedicated path around the device's boundary (hence the name). The path creates a virtual access capability that circumvents the normal inputs and provides direct control of the device and detailed visibility at its outputs.[3] The contents of the boundary scan are usually described by the manufacturer using a part-specific BSDL file.

Among other things, a BSDL file will describe each digital signal exposed through pin or ball (depending on the chip packaging) exposed in the boundary scan, as part of its definition of the Boundary Scan Register (BSR). A description for two balls might look like this:

```
"541 (bc_1,          *,  control,  1)," &
"542 (bc_1,     GPIO51_ATACS1,  output3,  X,    541, 1, Z)," &
"543 (bc_1,     GPIO51_ATACS1,    input, X)," &
"544 (bc_1,          *,  control,  1)," &
"545 (bc_1,     GPIO50_ATACS0,  output3,  X,    544, 1, Z)," &
"546 (bc_1,     GPIO50_ATACS0,    input, X)," &
```

That shows two balls on a mid-size chip (the boundary scan includes about 620 such lines, in a 361-ball BGA package), each of which has three components in the BSR: a control configuring the ball (as input, output, what drive level, pullups, pulldowns, and so on); one type of output signal; and one type of input signal.

There are JTAG instructions to SAMPLE the data in that boundary scan register, or PRELOAD it with values.

During testing, I/O signals enter and leave the chip through the boundary-scan cells. Testing involves a number of test vectors, each of which drives some signals and then verifies that the responses are as expected. The boundary-scan cells can be configured to support external testing for interconnection between chips (EXTEST instruction) or internal testing for logic within the chip (INTEST instruction).

Board Test Infrastructure

Typically high-end commercial JTAG testing systems allow the import of design 'netlists' from CAD/EDA systems plus the BSDL models of boundary scan/JTAG complaint devices to automatically generate test applications. Common types of test include

- Scan-path 'infrastructure' or integrity
- Boundary-scan device pin to boundary-scan device pin 'interconnect'
- Boundary-scan pin to memory device or device cluster (SRAM, DRAM, DDR etc)
- Arbitrary logic cluster testing

When used during manufacturing, such systems also support non-test but affiliated applications such as in-system programming of various types of flash memory: NOR, NAND, and serial (I2C or SPI).

Such commercial systems are used by board test professionals and will often cost several thousand dollars for a fully-fledged system. They can include diagnostic options to accurately pin-point faults such as open circuits and shorts and may also offer schematic or layout viewers to depict the fault in a graphical manner. Tests developed with such tools are frequently combined with other test systems such as in-circuit testers (ICTs) or functional board test systems.

UNIT V SPECIFICATION USING VERILOG HDL

Basic concepts

Verilog HDL:

Verilog is a *HARDWARE DESCRIPTION LANGUAGE (HDL)*. A hardware description Language is a language used to describe a digital system, for example, a microprocessor or a memory or simple flip-flop. This just means that by using a HDL one can describe any hardware (digital) at any level.

White space:

White space characters, such as SPACE and TAB and blank lines are ignored by verilog compiler. Multiple statements are written on a single line such as,

```
f = a0; if (s == 0) f = a1;
```

Placing each statement on a separate line and using indentation within blocks of code, such as an if-else statement are good ways to increase the readability of code.

Documentation in verilog code:

Documentation can be included in verilog code by writing a comment. A short comment begins with double slash, //, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters /* and */.

```
//this is a short comment
/* this is a long comment
   That spans two lines */
```

Operators:

Verilog has operators of three types. They are *unary operators* which precede the operand, *binary operators* which appear between two operands and *ternary operators* that have two separate operators that separate three operands.

Example:

```
x = + y; //+ is unary operator, y is the operand
x = y && z; // && is a binary operator, y and z are operands
x = a ? b : c; // ? : is a ternary operator, a, b, c are operands
```

Signal values, Numbers:

Verilog supports scalar nets and variables that can be represent individual signals and vectors that correspond to multiple signals. Each individual signal can have four possible values:

0 = logic value 0;

1 = logic value 1;

z = high impedance;

x = unknown value;

The value of a vector variable is specified by giving a constant of the form

[size][’radix]constant

where size is the number of bits in the constant and resix is the number base.

Radices are:

d = decimal;

b = binary;

h = hexadecimal;

o = octal;

when no radix is specified the default is decimal.

Constants are:

0	number 0;
10	decimal number 10;
’b10	binary number 10 = $(2)_{10}$;
’h10	hex number 10 = $(16)_{10}$;
4’b100	binary number 0100 = $(4)_{10}$;
4’bx	unknown 4 bit value xxxx;
8’hfx	equivalent to 8’b1111_xxxx;

Parameters:

A *parameter* in verilog associates an identifier name with a constant.

Declaration:

parameter n = 4;

parameter s0 = 2’b00, s1 = 2’b 01; s2 =2’b11; s3 = 2’b 10;

Nets:

A *net* represents a node in a circuit. There are two different types of nets are used in verilog. They are *wire* and *tri*.

Wire type net can be employed to connect an output of one logic element in a circuit to an input of another logic element. Examples of *scalar* wire declarations:

wire a;

wire a, b;

Example of *vector* wire declaration:

```
wire [2:0] s;
wire [1:3] x;
```

The square brackets are the syntax for specifying a vector's range. The range [$R_{a:}R_b$] can be either increasing or decreasing. The indices R_a and R_b can be either positive or negative integers.

The net s can be used as a three bit quantity or each bit can be referred to individually as s[2], s[1] and s[0]. If a value assigned to s such as s = 3'b010, the result is s[2] = 0, s[2] = 1 and s[0] = 0.

The tri type denotes circuit nodes that are connected in a tri state fashion.

Example:

```
tri x;
tri [7:0] out;
```

These nets are treated in the same manner as the wire type and they are used only to enhance the readability of code includes tri state gates.

Variables:

A variable can be assigned a value in one verilog statement and it retains this value until it is overwritten in a subsequent assignment statement. There are two types of variables, reg and integer.

```
reg [2:0] out;
```

The above declarations show that out has three bits. The keyword reg does not denote a storage element or register. In verilog code *reg* variables can be used to model either combinational or sequential parts of a circuit.

```
integer x;
```

Integer variables are useful for describing the behaviour of a module, but they do not directly correspond to nodes in a circuit.

Memories:

Memory is a two dimensional array of bits. Verilog allows such a structure to be declared as a variable (reg or integer) that is an array of vectors, such as

```
reg[3:0] x [1:0]
```

This statement defines x as two four-bit variables named x[1] and x[0]. Memories cannot be net types and they cannot be used as ports on a module.

Identifiers

Identifiers are the names of variables and other elements in verilog code.

Rules:

a) Any letter or digit may be used

b) The _underscore and $ characters are used

c) Identifiers must not begin with a digit and it should not be a verilog keyword

Example: legal identifiers are a, x1, a_b and byte.

Illegal identifiers are 1a, +b, a*b and 258.

Verilog is case sensitive, hence k is not the same as K and BYTE is not the same as Byte.

Verilog allows a second form of identifier called an escaped identifier. These identifiers are begin with (\) backslash character, which can then be followed by any printable ASCII characters except white spaces.

Example: \234, \sig-name, \x+y

Escaped identifiers should not be used in normal verilog code, they are intended for use in code produced automatically when other languages are translated into verilog.

Structural level modeling:

In gate level modeling or structural level modeling, the circuit is described in terms of logic gates. Example: and, or, nand, nor, etc... The logic gates have one scalar output and multiple scalar inputs. The first terminal in the list of gate terminals is an output and the other terminals are inputs.

a. Gate Primitives

Verilog defines some basic logic gates as part of the language. Gate primitives used in verilog codes are NOT, AND, OR, NAND, NOR, XOR, XNOR gate. Verilog includes predefined modules that implement basic logic gates. These gates allow a circuit's structure to be described using *gate instantiation* statements of the form:

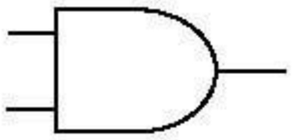
```
gate_name [instance_name] (output_port, input_port {, input_port});
```

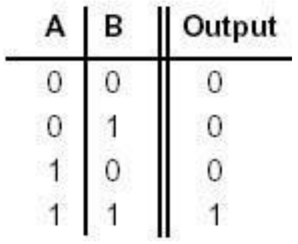
Here *gate_name* specifies the gate type and *instance_name* is an identifier, also it is optional.

Each gate may have different number of ports, the output port listed first, followed by a variable number of input ports.

Example:

AND

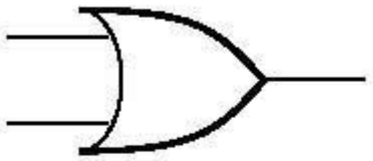




A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

```
and And1 (output, a, b);
```

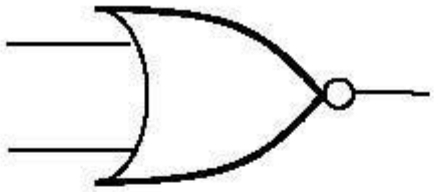
OR



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

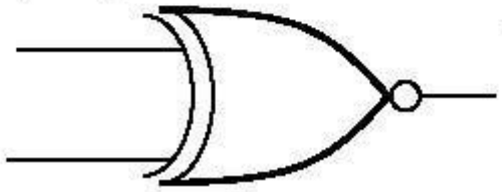
```
or Or1 (output, a, b);
```

NOR



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

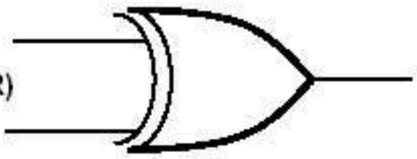
Exclusive NOR (XNOR)



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

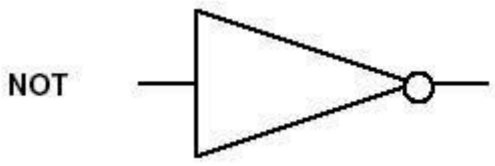
```
nor Nor1 (output, a, b);

xnor Xnor1 (output, a, b);
```

Exclusive OR (XOR)



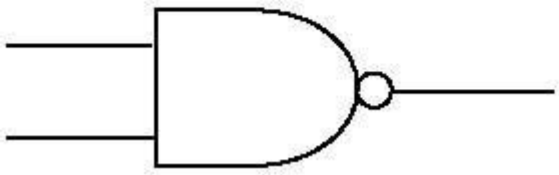
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

```
xor Xor1 (output, a, b);
```



Input	Output
0	1
1	0

```
not Not1 (output, input);
```



NAND:

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

```
nand Nand (output, a, b);
```

Structural specification of a full adder:

```
module fulladd (a,b,cin, sum,carry) ;

input a, b, cin;

output sum, carry;

wire p, q, r;

and And1 (p, a, b);

and And2 (q, a, cin);

and And3 (r, b, cin);

or Or1 (carry, p, q, r);

xor Xor1 (sum, a, b, cin);

endmodule
```

Simplified version of full adder:

```
module fulladd (a,b,cin, sum,carry) ;
input a, b, cin;
output sum, carry;
wire p, q, r;
and (p, a, b);
and (q, a, cin);
and (r, b, cin);
or (carry, p, q, r);
xor (sum, a, b, cin);
endmodule
```

Gate delays

A delay, as used in Verilog, is a number of time units it takes to receive a response of a circuit. In a simple forward combinational circuit this is a time it takes to obtain a change on an output when an input is altered. Let's say a synchronous circuit is triggered by a positive clock transition; `*the delay'* is then the time it takes for the circuit to change its outputs from the time the clock line is set high rather then a time it takes to change the outputs since data lines altered.

Types of delays

There are three basic types of delay which can be used:

- the time it takes to set an output high (t_{rise}); this applies to a transition which may start from any (`0', `1', `X' or `Z') state
- the time it takes to set an output low (t_{fall}); similarly, applies to transitions which begin in any state.
- and finaly, the time it takes to cut-off a buffer (t_{off}).

Syntax gate with delay:

```
gate_type #(t_rise, t_fall, t_off) instance_name (output_port,input_port1, input_port2,...);
```

For example:

```
and #(1, 3) g1 (o1, i1, i2);

nor #(2) g2 (o2, i3, i4);
```

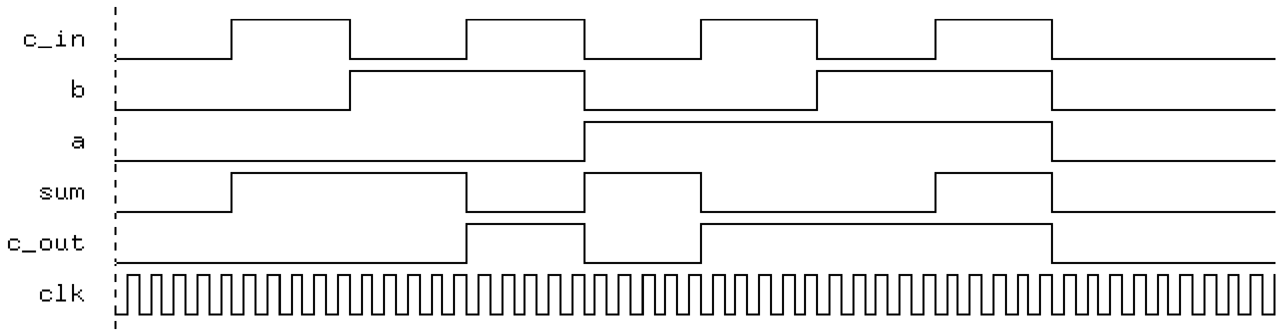
The above parameters are specified in sequence and if a cut-off time is to be specified the fall time *cannot* be omitted. One more interesting transition is a move of an input from a defined logic state i.e. either `0' or `1' to an `X' (don't care). In such case, the shortest of the two times $t_{fall}$, $t_{rise}$ is used. For an inverse case (`X'->`0'|`1') the longer delay is used.

Example: Single bit full-adder

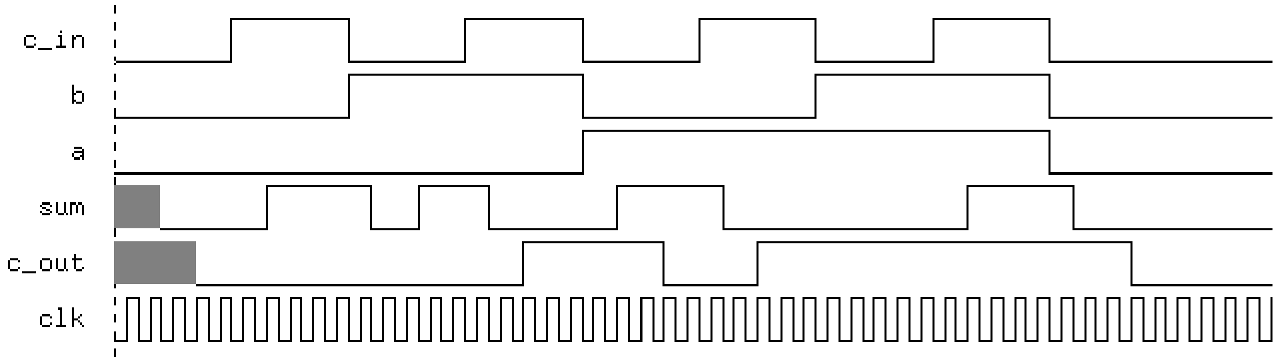
The design below is for a full adder, written using gate-level modelling techniques.

A gate-level model of a 1-bit full-adder:

```
module full_adder(sum, c_out, a, b, c_in);
output c_out, sum;
input a, b, c_in;
wire p, q, r;
xor #(3,2) x0 (p, a, b);
xor #(3,2) x1 (sum, p, c_in);
and #(2,4) a0 (q, a, b);
and #(2,4) a1 (r, p, c_in);
or #(3) o0 (c_out, q, r);
endmodule
```



Full Adder: Output - no delays



Full Adder: Output - delays as specified

Operators

Verilog has large number of operators.

Binary Arithmetic Operators

Binary arithmetic operators operate on two operands. Register and net (wire) operands are treated as unsigned. However, real and integer operands may be signed. If any bit is unknown ('**x**') then result is unknown.

Operator	**Name**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Relational Operators

Relational operators compare two operands and return a logical value, i. e., TRUE(1) or FALSE(0). If any bit is unknown, the relation is ambiguous and the result is unknown.

Operator	**Name**
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Logical equality
!=	Logical inequality

Logical Operators

Logical operators operate on logical operands and return a logical value, i. e., TRUE(1) or FALSE(0). Used typically in **if** and **while** statements. Do not confuse logical operators with the bitwise Boolean operators. For example , ! is a logical NOT and ~ is a bitwise NOT. The first negates, e. g., !(5 == 6) is TRUE. The second complements the bits, e. g., ~{1,0,1,1} is 0100.

Operator	**Name**
!	Logical negation
&&	Logical AND
||	Logical OR

Bitwise Operators

Bitwise operators operate on the bits of the operand or operands. For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (**x**) bit results in the expected value. For example, the AND of an **x** with a FALSE is an **x**. The OR of an **x** with a TRUE is a TRUE.

Operator	**Name**
~	Bitwise negation
&	Bitwise AND
|	Bitwise OR

^	Bitwise XOR
~&	Bitwise NAND
~|	Bitwise NOR
~^ or ^~	Bitwise NOT XOR

Unary Reduction Operators

Unary reduction operators produce a single bit result from applying the operator to all of the bits of the operand. For example, **&A** will **AND** all the bits of **A**.

Operator	**Name**
&	AND reduction
|	OR reduction
^	XOR reduction
~&	NAND reduction
~|	NOR reduction
~^	XNOR reduction

Unary Arithmetic Operators

Operator	**Name**	**Comments**
-	Unary Minus	Changes sign of its operand.

Timing controls

A timing control is either a delay control or an event control.

Delay control:

A delay control delays an assignment by a specified amount of time. A timescale compiler directive is used to specify the units of time followed by the precision used to calculate time expressions,

```
`timescale 1ns/10ps // Units of time are ns. Round times to 10 ps.
```

Time units may only be s , ns , ps , or fs and the multiplier must be 1, 10, or 100. We can delay an assignment in two different ways:

- Sample the RHS immediately and then delay the assignment to the LHS.
- Wait for a specified time and then assign the value of the LHS to the RHS.

Here is an example of the first alternative (an intra-assignment delay):

```
a = #1 b; // intra-assignment delay
```

The second alternative is delayed assignment:

```
#1 a = b; // delayed assignment
```

These two alternatives are not the same. The intra-assignment delay is equivalent to the following code:

```
begin // Equivalent to intra-assignment delay.
hold = b; // Sample and hold y immediately.
#1; // Delay.
a = hold; // Assignment to a. Overall same as a = #1 b.
end
```

In contrast, the delayed assignment is equivalent to a delay followed by an assignment as follows:

```
begin // Equivalent to delayed assignment.
#1; // Delay.
a = b; // Assign y to x. Overall same as #1 a = b.
end
```

Event control:

An event control, delays an assignment until a specified event occurs. Here is the formal definition:

```
event_control ::= @ event_identifier | @ (event_expression)
event_expression ::= expression | event_identifier
| posedge expression | negedge expression
| event_expression or event_expression
```

A positive edge (denoted by the keyword posedge) is a transition from '0' to '1' or 'x' , or a transition from 'x' to '1 '. A negative edge (negedge) is a transition from '1' to '0' or 'x' , or a transition from 'x' to '0'. Transitions to or from 'z' do not count. Here are examples of event controls:

```
module delay_controls; reg X, Y, Clk, Dummy;
always #1 Dummy=!Dummy; // Dummy clock, just for graphics.
// Examples of delay controls:
```

```
always begin #25 X=1;#10 X=0;#5; end
// An event control:
always @(posedge Clk) Y=X; // Wait for +ve clock edge.
always #10 Clk = !Clk; // The real clock.
initial begin Clk = 0;
$display("T  Clk X Y");
$monitor("%2g",$time,,,Clk,,,,X,,Y);
$dumpvars;#100 $finish; end
endmodule
```

```
T  Clk X Y
0 0   x x
10  1   x x
20  0  x  x
25  0  1  x
30  1  1  1
35  1  0  1
40  0  0  1
50  1  0  0
60  0  0  0
65  0  1  0
70  1  1  1
75  1  0  1
80  0  0  1
90  1   0 0
```

The dummy clock in delay_controls helps in the graphical waveform display of the results (it provides a one-time-tick timing grid when we zoom in, for example). Figure shows the graphical output (white is used to represent the initial unknown values). The assignment statements to 'X' in the always statement repeat (every 25 + 10 + 5 = 40 time ticks).

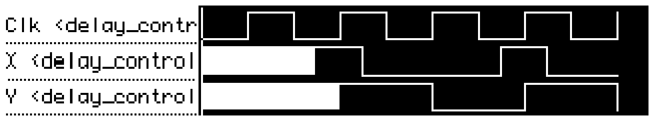


FIGURE: Output from the module delay_controls .

Events can be declared (as named events), triggered, and detected as follows:

```
module show_event;
reg clock;
event event_1, event_2; // Declare two named events.
```

```
always @(posedge clock) -> event_1; // Trigger event_1.
always @ event_1
begin $display("Strike 1!!"); -> event_2; end // Trigger event_2.
always @ event_2 begin $display("Strike 2!!");
$finish; end // Stop on detection of event_2.
always #10 clock = ~ clock; // We need a clock.
initial clock = 0;
endmodule
```

Procedural assignments conditional statements

Verilog provides procedural statements. Procedural statements are executed in which order they appear in the code. Verilog syntax requires that procedural statements be contained inside an always block.

Always blocks:

An always block is a construct that contains one or more procedural statements. It has the form:

```
always @ (sensitivity_list)
[begin]
[procedural assignment statements]
[if-else statements]
[case statements]
[while, repeat, and for loops]
[task and function calls]
[end]
```

Verilog includes several types of procedural statements. When multiple statements are included in an always block, the *begin* and *end* keywords are used, otherwise it can be omitted.

The sensitivity_list is a list of signals that directly affect the output results generated by the always block.

Example for always block:

```
always @ (a or b)
begin
        sum = a ^ b;
        carry = a & b;
end
```

Since the output variables *sum* and *carry* depend on *a* and *b*, these signals are included in the sensitivity list, separated by a keyword *or*. If the value of a signal in the sensitivity list changes, then the statements inside the *always* block are evaluated in the order presented.

Procedural assignment statements:

Any signal assigned a value inside an always block has to be a variable of type *reg* or *integer*. A value is assigned to a variable with a procedural assignment statement. There are two types of assignments:

a) blocking assignments
b) non-blocking assignments

Blocking assignments are denoted by the symbol (=).

The term blocking means that the assignment statement completes and updates its left – hand side before the subsequent statement is evaluated.

Example:

```
sum = a + b;
x = sum [0];
```

The first statement sets *sum* using the current values of *a* and *b*, and then the second statement sets *p* according to this new value of *sum*.

Verilog also provides non-blocking assignments and it can be denoted by (<=).

Example:

```
s <= a + b;
p <= s [0];
```

At simulation time t_i, the statements are still evaluated in order, but they both use the values of variables that exist at the start of the simulation time, t_i.

The first statement determines a new value for *s* based on the current values of *a* and *b*, but *s* is not actually changed to this value until all statements in the associated *always* block have been evaluated. So, the value of *p* at time t_i is based on the value of *s* at time $t_{i-1.}$

Summary:

For blocking assignments, the values of variables seen at time t_i by each statement are the new values set in t_i by any preceding statements in the *always* block.

For non-blocking assignments, the values of variables seen at time t_i are the values set in time t_{i-1}.

The if-else statement:

General form of the if-else statement is given in figure.

```
if (expression 1)
  begin
          statement;
  end
else if (expression 2)
  begin
          statement;
  end
else
  begin
        statement;
  end
```

Figure

If expression 1 is true, then the first statement is evaluated. When multiple statements are involved, they have to be included inside a begin-end block.

Example:

Verilog code for 2 x 1 multiplexer

```
always @ (a or b or s)
  if (s == 0)
           f = a;
  else
           f = b;
```

The code defines a 2 x 1 multiplexer with data inputs *a* and *b*, with select input *s* and output *f*.

Case statement:

General form of case statement is shown in the figure.

```
case (expression)
            alternative1: begin
                                      statement:
                                    end
            alternative2 : begin
                                      statement;
                                    end
                 [default:  begin
                                      statement;
                                    end]
endcase
```

The bits in the expression are called *controlling expression*, it checked for a match with each alternative. The first successive match causes the associated statements to be evaluated. Each digit in each alternative is compared for a correct match of the four values 0, 1, x and z.

A special case is the *default* clause, which takes effort if no other alternative matches.

Example:

Verilog code for 2 x 1 multiplexer.

```
always @ (x or y or s)
        case (s)
                1'b0: f = x;
                1'b1: f = y;
        endcase
```

In this example, *s* can have four values *0, 1, x, z.* If s = 0 means output f = x, and if s = 1 means output f = y.

Example: Verilog code for full adder using *case* statement.

```
module fulladd (a, b, cin, sum, carry);

input a, b, cin;

output sum, carry;

reg sum, carry;

always @ (a or b or cin)

begin

          case ({a, b, cin})

          3’b000: {carry, sum} = ’b00;                    3

          3’b001: {carry, sum} = ’b01;

          3’b010: {carry, sum} = ’b10;

          3’b011: {carry, sum} = ’b01;

          3’b100: {carry, sum} = ’b10;

          3’b101: {carry, sum} = ’b10;

          3’b110: {carry, sum} = ’b11;

          3’b111: {carry, sum} = ’b00;

endcase

end

endmodule
```

Loop statements:

Verilog code includes four types of loop statements: *for* loop, *while* loop, *repeat* loop and *forever* loop. Synthesis tool support the *for loop*.

General form of *for loop* is given below.

```
for (initial_index; terminal_index; increment)

begin

      statement;

end
```

The initial_index is evaluated once, before the first loop iteration and typically performs the initialization of the integer loop control variable. In each loop iteration, the begin-end block is performed. The terminal_index condition is checked and if it is true, then another loop iteration is done.

Example:

Verilog code for bit-counting using *for* loop.

```
always @ (x)

begin

            count=0;

            for (x = 0; x < n; x = x + 1)

            count = count + b (x);

end
```

General form of *while* loop and *repeat* loop is given below.

```
while (condition)

begin

            statement;

end
```

```
repeat (constant_value)

begin

            statement;

end
```

Tasks and Functions

Tasks are like procedures in other programming languages, e. g., tasks may have zero or more arguments and do not return a value. Functions act like function subprograms in other languages. Except:

1. A Verilog function must execute during one simulation time unit. That is, no time controlling statements, i. e., no delay control (#), no event control (@) or **wait** statements, allowed. A task can contain time controlled statements.

2. A Verilog function cannot invoke (call, enable) a task; whereas a task may call other tasks and functions.

The definition of a task is the following:

```
task <task name>;  // Notice: no list inside ()s
            <argument ports>
            <declarations>
            <statements>
endtask
```

An invocation of a task is of the following form:

```
<name of task> (<port list>);
```

where **<port list>** is a list of expressions which correspond to the **<argument ports>** of the definition. Port arguments in the definition may be **input**, **inout** or **output**. Since the **<argument ports>** in the task definition look like declarations, the programmer must be careful in adding declares at the beginning of a task.

Testing tasks and functions

```
module tasks;
task add;      // task definition
 input x, y; // two input argument ports
 output z; // one output argument port
 reg R;  // register declaration
 begin
   R = 1;
   if (x == y)
   z = 1 & R;
   else
     z = 0;
 end
endtask

initial begin: init1
   reg p;
   add(1, 0, p);  // invocation of task with 3 arguments
   $display("p= %b", p);
end
endmodule
```

The parameters **input** and **inout** are passed by value to the task and **output** and **inout** parameters are passed back to invocation by value on return. Call by reference is not available.

Allocation of all variables is static. Therefore, a task may call itself but each invocation of the task uses the same storage, i. e., the local variables are *not* pushed on a stack. Since concurrent threads may invoke the same task, the programmer must be aware of the static nature of storage and avoid unwanted overwriting of shared storage space.

The purpose of a function is to return a value that is to be used in an expression. A function definition must contain at least one **input** argument. The passing of arguments in functions is the same as with tasks (see above). The definition of a function is the following:

```
function <range or type> <function name>;
        <argument ports>
        <declarations>
        <statements>
```

```
            endfunction
```

where **<range or type>** is the type of the results passed back to the expression where the function was called. Inside the function, one must assign the function name a value. Below is a function which is similar to the task above.

Testing functions

```
module functions;
function [1:1] add2; // function definition
 input x, y;         // two input argument ports
 reg R;              // register declaration
 begin
   R = 1;
   if (x == y)
   add2 = 1 & R;
   else
     add2 = 0;
end
endfunction
initial begin: init1
   reg p;
   p = add2(1, 0); // invocation of function with 2 arguments
   $display("p= %b", p);
end
endmodule
```

Data flow modeling

In verilog coding gate-level modeling works well due to the number of gates are less; if large number of gates are used in a circuit then this type of modeling will be complicated. Dataflow modeling is a powerful approach to implement large circuit. This modeling becomes a popular approach as logic synthesis tools have become difficult.

Continuous assignments:

Continuous assignments are one type of *concurrent* statements. While gate instantiations allow the description of a circuit’s structure, continuous assignments allow the description of a circuit’s function.

General form of the *continuous statement* is

```
assign net_assignment {, net_assignment};
```

the net_assignment can be any expression involving the operators.

Example:

```
assign carry = (a & b) | (a & c) | (b & c);
assign sum = a ^ b ^ c;
```

multiple assignments are specified in one assignment, using commas to separate the assignment.

Example:

```
assign carry = (a & b) | (a & c) | (b & c), sum = a ^ b ^ c;
```

Example for multibit assignment is

```
wire [1:3] x, y, z;
                .
                .
                .
assign z = x & y;
```

This results in $z_1 = x_1y_1$, $z_2 = x_2y_2$, $z_3 = x_3y_3$.

Delays in dataflow modeling:

Delay values control the time between the change in a right hand side operand and when the new value is assigned to the left hand side. Delays in continuous assignment statements can be specified in three ways:

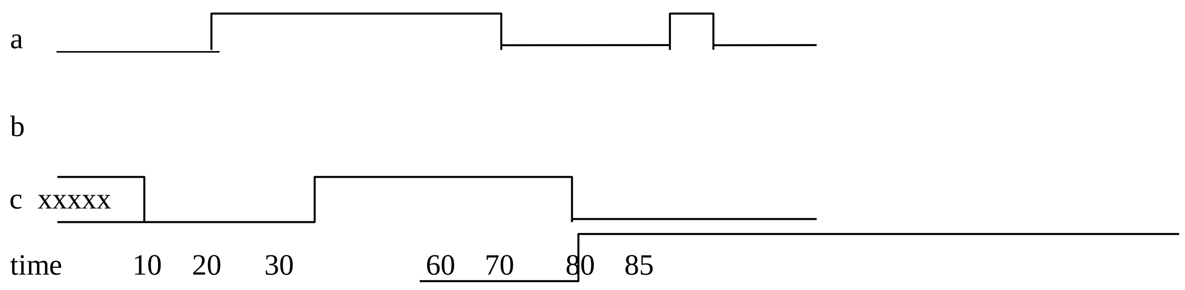
a) Regular assignment delay
b) Implicit continuous assignment delay
c) Net declaration delay

Regular assignment delay:

In this way, the delay value can be specified after the keyword *assign.*

Example:

```
assign #10 c = a & b;
```



1. When the input *a* and *b* go to high at time 20, *c* goes to high at 10 time units later.

2. When *a* goes low at 60, *c* goes to low at 70.
3. When *a* changes to high at 80, but it goes down to low before 10 time units have over and done.
4. So, at the time of recomputation, 10 units after 80, a is 0. Thus, *c* gets the value 0.

Implicit continuous assignment delay:

Implicit continuous assignment method is used to specify both a delay and an assignment on the *net*.

Example:

```
wire c;

assign #10 c = a & b;
```

In implicit continuous assignment delay we can write the above as:

```
wire #10 c = a & b;
```

Net declaration delay:

Net declaration delays can be used in gate level modeling. Delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net *c*, then any value change applied to the net *c* is delayed accordingly.

Example:

```
wire c;

assign #10 c = a & b;
```

We can write the above by using net declaration delay as:

```
wire #10 c;

assign c = a & b;
```

Switch level modelling:

This is another type of modeling used in verilog HDL. In this modeling, MOS switches are used. In verilog HDL, transistors are also known as switches that either conduct or are open.

MOS Switches

Models of MOS networks consist following four primitive types:

nmos

pmos

rnmos

rpmos

The *pmos* keyword stands for PMOS transistor and the *nmos* keyword stands for NMOS transistor. PMOS and NMOS transistors have relatively low impedance between their sources and drains when they conduct.

The *rpmos* keyword stands for resistive PMOS transistor and the *rnmos* keyword stands for resistive NMOS transistor. Resistive PMOS and resistive NMOS transistors have significantly higher impedance between their sources and drains when they conduct than PMOS and NMOS transistors have. These four gate types are unidirectional channels for data similar to the bufif gates.

Declarations of these gates begin with one of the following *keywords*:

pmos

nmos

rpmos

rnmos

The delay specification follows the *keyword*. A terminal list completes the declaration. The delay specification can be 0, 1, 2, or 3 delays. If there is no delay, there is no delay through the switch. A single delay determines the delay of all output transitions. If the specification contains 2 delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the 2 delays specifies the delay of transitions to Z and X. If there are 3 delays, the first delay specifies the rise delay, the second delay specifies the fall delay, the third delay determines the delay of transitions to Z, and the smallest of the three delays applies to transitions to X. Delays on transitions to H and L are the same as delays on transitions to X.

These four switches have one output, one data input, and one control input. The first terminal in the terminal list connects to the output, the second terminal connects to the data input, and the third terminal connects to the control input.

Some combinations of data input values and control input values cause these switches to output either of two values, without a preference for either value. These switches' logic tables include 2 symbols representing such unknown results. The symbol L represents a result which has a value of 0 or Z. The symbol H represents a result which has a value of 1 or Z.

pmos rpmos	CONTROL 0 1 x z	pmos rpmos	CONTROL 0 1 x z

D 0	0 z L L	D 0	0 z L L
A 1	1 z H H	A 1	1 z H H
T x	x z x x	T x	x z x x
A z	z z z z	A z	z z z z

Logic tables for **pmos, rpmos, nmos,** ***and*** **rnmos** *gates*

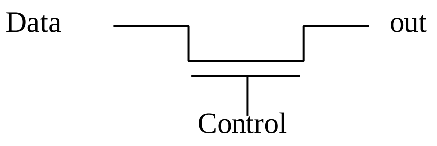
The following example declares a pmos switch:

pmos (out, data, control);

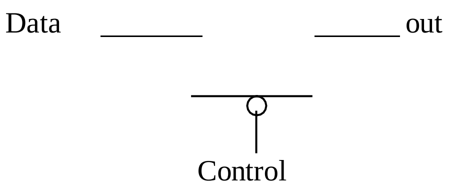
The output is out, the data input is data, and the control input is control.

Logic symbol:

Nmos switch:



Pmos switch:



Bidirectional Pass Switches

Declarations of bidirectional switches begin with one of the following keywords:

tran

tranif1

tranif0

rtran

rtranif1

rtranif0

Delay specification follows the *keywords* in declarations of *tranif1*, *tranif0*, *rtranif1*, and *rtranif0*; the next item is the optional identifier. A terminal list completes the declaration.

The delay specifications for tranif1, tranif0, rtranif1, and rtranif0 devices can be 0, 1, or 2 delays. If there is no delay, the device has no turn-on or turn-off delay. If the specification contains one delay, that delay determines both turn-on and turn-off delays. If
there are 2 delays, the first delay specifies the turn-on delay, and the second delay specifies the turn-off delay.

These six devices do not delay signals propagating through them. When these devices are turned off they block signals, and when they are turned on they pass signals.

The tranif1, tranif0, rtranif1, and rtranif0 devices have three items in their terminal lists. Two are bidirectional terminals that conduct signals to and from the devices, and the other terminal connects to a control input. The terminals connected to inouts precede the
terminal connected to the control input in the terminal list.

The tran and rtran devices have terminal lists containing two bidirectional terminals.
The bidirectional terminals of all six of these devices connect only to scalar nets or bit-selects of expanded vector nets.

The following example declares a *tranif1*:

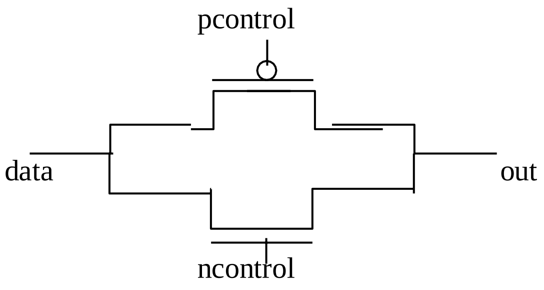
tranif1 (inout1, inout2, control);

The bidirectional terminals are *inout1* and *inout2*. The control input is *control*.

CMOS Gates

The cmos gate is the combination of a pmos gate and an nmos gate. The rcmos (resistive cmos) gate is the combination of an rpmos gate and an rnmos gate. The combined gates in these configurations share data input and data output terminals, but they have separate control inputs.

Cmos switch:



Declarations of these gates begin with one of these keywords:

cmos

rcmos

The delay specification can be 0, 1, 2 or 3 delays. If there is no delay, there is no delay through the gate. A single delay specifies the delay for all transitions. If the specification contains two delays, the first delay determines the rise delay, the second delay determines the fall delay, and the smaller of the two delays is the delay of transitions to Z
and X. If the specification contains three delays, the first delay controls rise delays, the second delay controls fall delays, the third delay controls transitions to Z, and the smallest of the three delays applies to transitions to X. Delays in transitions to H or L are the same as delays in transitions to X.

The cmos and rcmos gates have a data input, a data output and 2 control inputs. In the terminal list, the first terminal connects to the data output, the second connects to the data input, the third connects to the n-channel control input and the last connects to the p-channel control input.

The equivalence of the cmos gate to the pairing of an nmos gate and a pmos gate is detailed in the following explanation:

```
cmos (w, datain, ncontrol, pcontrol);
```

is equivalent to:

```
nmos (w, datain, ncontrol);
pmos (w, datain, pcontrol);
```

pullup and pulldown Sources

Declarations of these sources begin with one of the following keywords:

```
pullup
pulldown
```

A strength specification follows the keyword and an optional identifier follows the strength specification. A terminal list completes the declaration.

A pullup source places a logic value of one on the nets listed in its terminal list. A pulldown source places a logic value of zero on the nets listed in its terminal list. The signals that these sources place on nets have pull strength in the absence of a strength specification. There are no delay specifications for these sources because the signals they place on nets continue throughout simulation without variation.

The following example declares two pullup instances:

```
pullup (strong1, strong0)(neta),(netb);
```

In this example, one gate instance drives neta, the other drives netb.

Behavioural modeling

In this section we are going to discuss about the behavioural features of Verilog.

Modules

A circuit or subcircuit can be explained in verilog is called a module. The general structure of a module declaration is given below:

```
module module_name [(port_name {, port_name})];

[parameter declarations]

[input declarations]

[output declarations]

[inout declarations]
```

```
[wire or tri declarations]

[reg or integer declarations]

[function or task declarations]

[assign continuous assignments]

[initial block]

[always blocks]

[gate instantiations]

[module instantiations]

endmodule
```

General form of a module

The module has name, module_name, which can be any valid identifier followed by a list of ports. The name port refers to an input or output connection in an electrical circuit.

The ports used in verilog code are:

a) input port
b) output port
c) inout port

these ports can be either scalar or vector.

Example:

```
input a,b,c;

input [2:0]a,b;

output s,c;

inout [5:0]q;

output [3:0]s;

wire  x,y;

wire [5:0]y;

reg [ 2:0]x;
```

All procedural statements occur in blocks that are defined inside modules. There are two kinds of procedural blocks: the initial block and the always block. Within each block, Verilog uses a begin and end to enclose the statements. Since initial blocks are ignored during synthesis, only always blocks are discussed. Always blocks usually take the following format:

```
always
begin
statement
 .....
end
```

where each statement is a procedural assignment line terminated by a semicolon.

Module Declaration

In the module declaration, the I/O ports of the circuit are declared. Each port has a name and a mode (in, out, and inout) as shown in the example below.

```
module ex (A, B, C, D, E);
    input A, B, C;
    output D;
   inout E;
   wire D, E;
    ...
   assign E = oe ? A : 1'bz;
   assign D = B & E;
   ...
   endmodule
```

The input and output ports defined in the module declaration called ex are the basic input and output I/O signals for the design. The inout port in Verilog is analogous to a bi-directional I/O pin on the device with the data flow for output versus input being controlled by the enable signal to the tristate buffer.

The preceding example describes E as a tristate buffer with a high-true output enable signal. If oe = 1, the value of signal A will be output on the pin represented by E. If oe = 0, then the buffer is in high impedance (Z) and any input value driven on the pin E (from the external logic) will be brought into the device and fed to the signal represented by D.

Verilog Assignments

There are two forms of assignment statements in the Verilog language:

- Continuous Assignments
- Procedural Assignments

Continuous Assignments

Continuous assignments are used to model combinatorial logic in a concise way. Both explicit and implicit continuous assignments are supported. Explicit continuous assignments are introduced by the **assign** keyword after the net has been separately declared. Implicit continuous assignments combine declaration and assignment.

Example of an explicit continuous assignment:

```
wire x;
.....
assign x = select ? b : a;
```

Example of an implicit continuous assignment:

```
wire temp_hold = a | b;
```

Note Continuous assignments are only allowed on wire and tri data types.

Procedural Assignments

Procedural assignments are used to assign values to variables declared as regs and are introduced by always blocks, tasks, and functions. Procedural assignments are usually used to model registers and FSMs.

Different statements can be used in a process:

- Variable and signal assignment
- If... else statement
- Case statement
- For loop statement
- Function and task call

The following sections provide examples of each of these statements.

if...else statement

If... else statements use true/false conditions to execute statements. If the expression evaluates to true, the first statement is executed. A block of multiple statements may be executed using begin and end keywords. If...else statements may be nested. The following example shows how a MUX can be described using an If...else statement.

Example MUX Description Using If... Else Statement

```
module mux4 (sel, p, q, r, s, y);
input [1:0] sel;
input [1:0] p, q, r, s;
output [1:0] y;
reg [1:0] y;

always @(sel or p or q or r or s)
  begin
    if (sel[1])
      if (sel[0])
      y = s;
      else
        y = r;
    else
      if (sel[0])
      y = q;
      else
        y = p;
    end
endmodule
```

Case statement

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The Case statement evaluates the branches in the order they are written. The first branch that evaluates to true is executed. If none of the branches match, the default branch is executed.

Note Do no use unbounded integers in case statements. Always bound integers to a specific number of bits, or results will be unpredictable.

Casez treats all z values in any bit position of the branch alternative as a don't care.

Casex treats all x and z values in any bit position of the branch alternative as a don't care.

The question mark (?) can be used as a "don't care" in any of the preceding case statements. The following example shows how a MUX can be described using a Case statement.

Example MUX Description Using Case Statement

```
module mux4 (s a, b, c, d, y);
input [1:0] s;
input [1:0] a, b, c, d;
output [1:0] y;
reg [1:0] y;

always @(s or a or b or c or d)
  begin
    case (s)
      2'b00:  y  =  a;
    2'b01:  y  =  b;
    2'b10:  y  =  c;
    default:y  =  d;
    endcase
  end
endmodule
```

The preceding Case statement will evaluate the values of the input s in priority order. To avoid priority processing, it is recommended that you use a parallel-case Verilog meta comment which will ensure parallel evaluation of the sel inputs as in the following.

Example:

```
always @(sel or a or b or c or d) //synthesis parallel_case
```

For and Repeat loops

When using always blocks, repetitive or bit slice structures can also be described using the "for" statement or the "repeat" statement.

The "for" statement is supported for:

- Constant bounds
- Stop test condition using operators <, <=, > or >=
- Next step computation falling in one of the following specifications:
  - *var* = *var* + step
  - *var* = *var* - step

(where *var* is the loop variable and *step* is a constant value).

The repeat statement is only supported for constant values. The following example shows the use of a For Loop.

Example For Loop Description

```
module countzeros (a, Count);
input [7:0] a;
output [2:0] Count;
reg [2:0] Count;
reg [2:0] Count_Aux;
integer i;

always @(a)
  begin
    Count_Aux =  3'b0;
  for (i = 0; i < 8; i = i+1)
  begin
      if (!a[i])
        Count_Aux = Count_Aux+1;
    end
  Count = Count_Aux;
  end
endmodule
```

Blocking Versus Non-Blocking Procedural Assignments

The # and @ time control statements delay execution of the statement following them until the specified event is evaluated as true. Use of blocking and non-blocking procedural assignments have time control built into their respective assignment statement.

The # delay is ignored for synthesis.

The syntax for a blocking procedural assignment is shown in the following example:

```
reg a;
a = #10 (b | c);
```

or

```
if (in1) out = 1'b0;
else out = in2;
```

As the name implies, these types of assignments block the current process from continuing to execute additional statements at the same time. These should mainly be used in simulation.

Non-blocking assignments, on the other hand, evaluate the expression when the statement executes, but allow other statements in the same process to execute as well at the same time. The variable change only occurs after the specified delay.

The syntax for a non-blocking procedural assignment is as follows:

```
variable <= @(posedge or negedge bit) expression;
```

The following shows an example of how to use a non-blocking procedural assignment.

```
if (in1) out <= 1'b1;
else out <= in2;
```

Test bench:

Test bench is another verilog code that creates a circuit involving the circuit to be tested. This code will send different inputs to the code under test and get the output and displays to check the accuracy.

Example:

The verilog code for *AND* gate is given below. It is an example for simple testbench.

```
module andgate_tb;
wire t_y;
reg t_a, t_b;
andgate my_gate( .a(t_a), .b(t_b), .y(t_y) );

initial
begin

$monitor(t_a, t_b, t_y);
t_a = 1'b0;
t_b = 1'b0;
#5
t_a = 1'b0;
t_b = 1'b1;
#5
t_a = 1'b1;
t_b = 1'b0;
#5
t_a = 1'b1;
t_b = 1'b1;

end
endmodule
```

Example:

Teset bench for *full adder:*

```
module adder_using_always ();
reg a, b;
reg s, c;
always @ (a or b)
begin
{c,s} = a + b;
end
initial begin
$monitor (" A = %b
B = %b CARRY = %b SUM = %b",a,b,c,s);
#10  a = 0; b = 0;
```

```
#10  a = 1;
#10  b = 1;
#10  a = 0;
#10  b = 0;
#10 $finish;
end
endmodule
```

Example:

Test bench for *D-latch*

```
module dlatch();
reg q;
reg d, en;
always @ (d or en)
if (en) begin
q = d;
end
initial begin
$monitor (" ENABLE = %b D = %b  Q = %b", en, d, q);
#1 en = 0;
#1 d = 1;
#1 en = 1;
#1  d = 0;
#1  d = 1;
#1  d = 0;
#1 en = 0;
```

```
#10 $finish;
end
endmodule
```

Example:

Test bench for *D-flip flop*

```
module dff ();
reg clk,reset,preset,d;
reg  q;
always @ (posedge clk or posedge rst or posedge preset)
if (reset) begin
q <= 0;
end else if (preset) begin
q <= 1;
end else begin
q <= d;
end
```

Testbench code here

```
initial begin
$monitor("CLK = %b RST = %b PRESET = %b D = %b Q =
%b",
clk,rst,preset,d,q);
clk    = 0;
#1 rst  = 0;
preset = 0;
d      = 0;
#1  rst = 1;
#2  rst = 0;
#2  preset = 1;
#2  preset = 0;
repeat (4) begin
#2  d      = ~d;
end
```

```
#2 $finish;
end
always
#1 clk = ~clk;
endmodule
```